Corso Assembly di base

http://xoomer.virgilio.it/ramsoft/

Sommario

Capitolo 1: Strumenti necessari per la sezione Assembly Base....3
Capitolo 2: Introduzione....8
Capitolo 3: La matematica del computer....14
Capitolo 4: Il sistema di numerazione binario....41
Capitolo 5: La logica del computer....65
Capitolo 6: Reti combinatorie per funzioni matematiche....91
Capitolo 7: L'architettura generale del computer....143
Capitolo 8: Struttura hardware della memoria....161
Capitolo 9: La memoria dal punto di vista del programmatore....195
Capitolo 10: Architettura interna della CPU....208
Capitolo 11: Il codice macchina e il codice Assembly....240
Capitolo 12: Struttura di un programma Assembly....276
Capitolo 13: Assembling & Linking....312
Capitolo 14: Disassembling....367
Capitolo 15: Libreria di procedure per l'I/O....386
Capitolo 16: Istruzioni per il trasferimento dati....422
Capitolo 17: Istruzioni aritmetiche....475
Capitolo 18: Istruzioni aritmetiche per i numeri BCD....524
Capitolo 19: Istruzioni logiche....559
Capitolo 20: Istruzioni per il trasferimento del controllo....603
Capitolo 21: Istruzioni per il controllo della CPU....627
Capitolo 22: Istruzioni per la manipolazione delle stringhe....630
Capitolo 23: Istruzioni varie....662
Capitolo 24: Operatori, direttive e macro....673
Capitolo 25: I sottoprogrammi....693
Capitolo 26: La ricorsione....730
Capitolo 27: I modelli di memoria....755
Capitolo 28: Linking tra moduli Assembly....799
Capitolo 29: Caratteristiche avanzate di MASM e TASM....832
Capitolo 30: Interfaccia tra C e Assembly....865
Capitolo 31: Interfaccia tra Pascal e Assembly....910
Capitolo 32: Interfaccia tra BASIC e Assembly....932

Appendici

Appendice A: Tabella dei codici ASCII estesi....959
Appendice B: Set di istruzioni della CPU....963

Capitolo 1 - Strumenti necessari per la sezione Assembly Base

La sezione **Assembly Base** si rivolge non solo a chi vuole imparare a programmare in **Assembly**, ma anche a chi non ha la minima nozione sul funzionamento di un computer; per seguire questo tutorial non e' richiesta quindi nessuna conoscenza di informatica o di hardware.

1.1 Il linguaggio Assembly

I computers si suddividono in una serie di categorie chiamate **famiglie hardware**; questa definizione e' legata all'architettura interna che caratterizza ogni elaboratore elettronico. Tutti i computers appartenenti alla stessa famiglia hardware, presentano una importantissima caratteristica chiamata **compatibilita'**; compatibilita' significa che tutti i computers appartenenti alla stessa famiglia hardware possono dialogare tra loro, il che equivale a dire che tutti i computers compatibili parlano lo stesso linguaggio. Il linguaggio parlato da un computer prende il nome di **machine code** (codice macchina o linguaggio macchina); come vedremo nei capitoli successivi, il codice macchina espresso in forma umana e' costituito da una sequenza di cifre ciascuna delle quali puo' assumere il valore **0** o **1**, e per questo motivo si parla anche di **codice binario**; dal punto di vista dei circuiti elettronici del computer, questi due valori rappresentano due possibili livelli di tensione assunti da un segnale elettrico. Possiamo associare ad esempio la tensione elettrica di **0** volt alla cifra **0**, e la tensione elettrica di **+5** volt alla cifra **1**. Come si puo' facilmente intuire, per un essere umano sarebbe una follia pensare di interagire con un computer attraverso il codice binario; si tenga presente comunque che all'inizio degli anni **50** il codice binario rappresentava l'unico mezzo per dialogare con gli elaboratori elettronici.
Proprio per risolvere questi problemi, si penso' allora di creare un apposito codice che risultasse piu' comprensibile agli esseri umani; nacque cosi' un codice chiamato **Assembly**. Il codice **Assembly** e' formato da una serie di mnemonici espressi da termini come **ADD**, **SUB**, **MUL**, **DIV**, **MOV**, etc; ciascuno di questi mnemonici corrisponde ad una ben precisa sequenza di cifre binarie, e chiaramente risulta molto piu' comprensibile del codice macchina. Il linguaggio macchina e il linguaggio **Assembly** vengono anche chiamati **low level languages** (linguaggi di basso livello); questa definizione si riferisce al fatto che questi linguaggi corrispondono a quello "grossolano" (o basso), parlato dal computer, in contrapposizione al linguaggio "raffinato" (o alto) parlato dagli esseri umani.
Ogni piattaforma hardware ha il proprio linguaggio **Assembly**; appare evidente quindi che i codici **Assembly** destinati ai computers appartenenti ad una determinata piattaforma hardware, risultano incomprensibili ai computers appartenenti ad un'altra piattaforma hardware. Per risolvere questo problema sono stati creati i cosiddetti **high level languages** (linguaggi di alto livello); i linguaggi di alto livello utilizzano una serie di mnemonici espressi da termini molto vicini al linguaggio umano (generalmente inglese), come ad esempio **IF**, **THEN**, **ELSE**, **WHILE**, **REPEAT**, **UNTIL**, etc. Un programma scritto con un linguaggio di alto livello viene prima tradotto nel codice macchina della piattaforma hardware in uso, e poi inviato alla fase di elaborazione; in questo modo e' possibile ottenere l'indipendenza di un programma dalla particolare piattaforma hardware che si sta utilizzando.
Da queste considerazioni si possono dedurre i vantaggi e gli svantaggi sia dei linguaggi di basso livello, sia dei linguaggi di alto livello. I linguaggi di basso livello presentano un grandissimo vantaggio rappresentato dalla capacita' di spremere al massimo tutto l'hardware disponibile nel computer; lo svantaggio principale e' rappresentato invece dal fatto di dipendere fortemente dalla piattaforma hardware in uso. I linguaggi di alto livello presentano il grandissimo vantaggio di risultare indipendenti dalle diverse piattaforme hardware; lo svantaggio principale e' dato invece dal fatto che, come vedremo nei successivi capitoli, questa indipendenza dall'hardware si paga (spesso a carissimo prezzo) in termini di prestazioni.

1.2 Ambiente operativo

Tutte le considerazioni esposte nella sezione **Assembly Base** e in tutte le altre sezioni di questo sito, si riferiscono ai computers appartenenti alla piattaforma hardware che viene indicata con il termine **PC i80x86**; questa sigla indica i cosiddetti **personal computers** basati sui **microprocessori Intel** o compatibili, della famiglia **80x86**; il microprocessore rappresenta il cervello pensante del computer e viene anche chiamato **CPU** o **Central Processing Unit** (unita' centrale di elaborazione). In passato questi computers venivano anche chiamati **IBM compatibili**; questa definizione e' legata al fatto che questa piattaforma hardware e' fortemente basata su tecnologie sviluppate inizialmente dalla **IBM**.
Il primo **PC** della storia e' stato il mitico **XT8086** realizzato dalla **IBM** nei primi anni **80**; questo nome deriva dal fatto che questo **PC** era basato sulla **CPU Intel 8086**. Successivamente sono stati realizzati **PC** basati su **CPU** sempre piu' potenti e sofisticate; si possono citare in ordine cronologico le **CPU 80286**, **80386** e **80486**. La generazione di **CPU** successiva all'**80486** viene indicata con la definizione di **classe Pentium**; questa classe viene anche indicata con le sigle **80586** e **80686**, e comprende le **CPU Athlon**, **Duron**, **Celeron**, etc.
Una caratteristica fondamentale delle **CPU 80x86** e' la cosiddetta **compatibilita' verso il basso**. Questa definizione indica il fatto che un programma scritto per un **8086**, funziona perfettamente su tutte le **CPU** di classe superiore; viceversa, un programma scritto espressamente per un **80586**, non funziona sulle **CPU** di classe inferiore.
Per ottenere la compatibilita' verso il basso, tutte le **CPU** della famiglia **80x86** sono in grado di simulare la modalita' di funzionamento dell'**8086**; questa modalita' viene chiamata **modalita' reale 8086**. Tutti i programmi di esempio presentati nella sezione **Assembly Base** si riferiscono a questa modalita' operativa delle **CPU 80x86**.
Si tenga presente che appena si accende il **PC**, tutte le **CPU** della famiglia **80x86** vengono inizializzate in modalita' reale; questa modalita' viene sfruttata dalle **CPU** di classe superiore per attivare altre modalita' operative piu' sofisticate, come la **modalita' protetta** che viene illustrata in una apposita sezione di questo sito.

Per permettere a qualunque utente di utilizzare un computer senza la necessita' di conoscerne il funzionamento interno, vengono realizzati appositi programmi chiamati **Sistemi Operativi** o **SO**; il **SO** di un computer puo' essere paragonato al cruscotto di una automobile. Sarebbe piuttosto difficile guidare una automobile senza avere a disposizione il volante, il freno, l'acceleratore, la spia dell'olio, etc; attraverso questi strumenti l'automobilista puo' gestire facilmente una automobile senza la necessita' di avere una laurea in ingegneria meccanica. Allo stesso modo, sarebbe praticamente impossibile utilizzare un computer senza avere a disposizione un **SO**; attraverso il **SO** anche un utente privo di qualsiasi conoscenza di informatica puo' ordinare al computer di suonare un CD audio, di aprire o di salvare un file, di ricevere o inviare la posta elettronica, etc.
Nell'ambiente operativo rappresentato dalla modalita' reale delle **CPU 80x86**, il **SO** piu' famoso e' sicuramente il **DOS** (Disk Operating System); questo **SO**, pur essendo ormai obsoleto, viene ancora supportato (o emulato) dai nuovi **SO** della famiglia **Windows**. In sostanza, anche in presenza di **Windows9x** o di **WindowsXP** e' possibile creare ed eseguire programmi destinati al **DOS**; esistono inoltre emulatori **DOS** che girano persino sotto **Linux**!
Il **DOS**, grazie alla sua struttura semplice e "trasparente", si presta molto bene per l'apprendimento dei concetti basilari del linguaggio **Assembly**; del resto, e' necessario tenere presente che le istruzioni fornite dal linguaggio **Assembly** dipendono, esclusivamente, dalla piattaforma hardware che si sta utilizzando, e non dal **SO**.
Bisogna anche aggiungere che, per un programmatore **Assembly**, e' fondamentale conoscere la **modalita' reale** supportata dal **DOS**; infatti, solo attraverso questa conoscenza e' possibile capire il perche' della particolare evoluzione che ha interessato le **CPU** della famiglia **80x86**.
Tutti i programmi di esempio presentati nella sezione **Assembly Base** sono espressamente destinati

al **SO DOS**; possiamo riassumere quindi tutte queste considerazioni dicendo che l'ambiente operativo di riferimento per la sezione **Assembly Base** e' quello rappresentato dalla modalita' reale **8086** supportata dal **DOS**.
In ambiente **Windows9x** esiste la possibilita' di riavviare il computer in modalita' **DOS**, oppure di lavorare da una finestra **DOS** chiamata anche **DOS Box** o **Prompt del DOS**; si tenga presente che la **DOS Box** di **Windows9x** fornisce un vero **DOS** e non un emulatore. Lo stesso **Windows9x** e' in realta' una sofisticata interfaccia grafica che si appoggia sul **DOS**; usando l'**Assembly** da una **DOS Box** e' possibile quindi fare di tutto, e se si commettono errori, si puo' anche mandare in crash **Windows9x**.
Il caso di **WindowsXP** invece e' differente in quanto questo **SO** ci mette a disposizione una **DOS Box** con un emulatore **DOS** la cui caratteristica principale e' l'estrema lentezza; questa lentezza e' dovuta ai vari "filtri protettivi" usati da **XP** per evitare che dalla **DOS Box** si possa violare la sicurezza del **SO**. Con **WindowsXP** e' possibile creare un floppy disk che permette di riavviare il **PC** in modalita' **DOS**; a tale proposito, si consiglia di consultare l'help in linea dello stesso **XP**. Si tenga anche presente che le future versioni di **Windows**, potrebbero anche abbandonare il supporto del **DOS**.
Da questo momento in poi, il termine **DOS** verra' utilizzato per indicare indifferentemente sia il **DOS** vero e proprio, sia la **DOS Box** di **Windows** (compresi gli emulatori); nel caso di programmi **DOS** incompatibili con **Windows**, verranno date tutte le necessarie informazioni.

1.3 Strumenti di sviluppo per l'Assembly

I moderni linguaggi di programmazione, vengono venduti insieme a sofisticati ambienti di sviluppo integrati chiamati **IDE** (Integrated Development Environment), che permettono di automatizzare tutto il lavoro che il programmatore deve svolgere; per poter imparare a sfruttare totalmente questi **IDE**, bisogna leggere manuali d'uso che spesso arrivano alle **500** pagine!
All'estremo opposto troviamo i programmatori **Assembly**, noti per la loro avversione nei confronti di queste (presunte) comodita'; d'altra parte si sa' che chi programma in **Assembly**, vuole avere il controllo totale sul computer e questo e' in netto contrasto con l'uso degli **IDE** troppo sofisticati che, essendo a tutti gli effetti dei programmi in esecuzione, potrebbero entrare in conflitto con il programma **Assembly** che stiamo testando. L'ideale quando si lavora in **Assembly** sarebbe avere il computer a completa disposizione del nostro programma. Con il passare del tempo, ci si abitua addirittura a lavorare con la linea di comando (**Prompt del DOS** per chi usa **Windows**) che consente di eseguire svariate operazioni ad altissima velocita' e con esigenze di memoria irrisorie.
Nel rispetto di questa filosofia, gli strumenti necessari per sviluppare programmi in **Assembly** sono pochissimi e semplici; la dotazione minima e' rappresentata dai seguenti strumenti:

*** un editor di testo**;
*** un assembler**;
*** un linker**.

L'**editor di testo** permette di creare, salvare, modificare o caricare dal disco file in formato ASCII; questa sigla sta per **American Standard Code for Information Interchange** (Codice Standard Americano per l'Interscambio dell'Informazione). Questo standard, definisce un insieme di **256** codici numerici riconosciuti dai **PC** di tutto il mondo; questo significa che ad esempio la sequenza di codici:

```
65 115 115 101 109 98 108 121
```

verra' tradotta e visualizzata da un qualsiasi editor di testo come la seguente sequenza di caratteri (stringa):

```
Assembly
```

Questo avverra' su qualunque **PC** e in qualunque nazione del mondo.

I primi **128** codici numerici (da **0** a **127**) sono uguali per tutti i **PC** mentre gli altri **128** (da **128** a **255**) variano da paese a paese con l'intento di rappresentare simboli caratteristici delle varie lingue; ad esempio la **n** spagnola con la tilde, la **c** francese di garcon etc.
I primi **32** codici numerici, sono chiamati "caratteri non stampabili" in quanto si tratta di codici di controllo, alcuni dei quali simulano il comportamento di una macchina da scrivere; ad esempio il codice **13** rappresenta il ritorno carrello (tasto **[Return]** o **[Invio]**), il codice **10** rappresenta l'avanzamento di una interlinea del cursore sullo schermo, il codice **9** rappresenta la tabulazione orizzontale (tasto **[TAB]**) e cosi' via. Ovviamente gli editor di testo devono essere in grado di interpretare correttamente anche questi codici.
In definitiva un file e' in formato ASCII quando aprendolo e visualizzandolo con un editor di testo, ci appare perfettamente leggibile; in contrapposizione ai file di testo troviamo invece i cosiddetti **file binari** che possono contenere programmi eseguibili, immagini, musica, etc. I file binari per essere visualizzati richiedono appositi programmi; se proviamo ad aprire un file binario con un editor di testo, ci troviamo davanti ad una sequenza incomprensibile di lettere dell'alfabeto, cifre, segni di punteggiatura, etc.
Un programma scritto in **Assembly** o in un qualsiasi linguaggio di programmazione, viene chiamato **codice sorgente** e deve essere rigorosamente in formato ASCII; questo significa che per leggere, scrivere, modificare e salvare un programma **Assembly** e' necessario usare un qualunque editor di testo. Utilizzando ad esempio strumenti come **Word** o **WordPad**, bisogna ricordarsi di salvare il file come documento di testo, mentre nel caso di **NotePad** (Blocco Note) o dell'editor del **DOS** (**EDIT.COM**) il formato testo e' quello predefinito. Si puo' scegliere l'editor piu' adatto ai propri gusti, tenendo presente che l'editor del **DOS** e' piu' che sufficiente; per avviare l'editor del **DOS** bisogna digitare **edit** dalla linea di comando.

L'**assembler** o assemblatore, e' un programma che traduce il codice sorgente in **codice oggetto** generando (se non ci sono errori) un file con estensione **.OBJ** chiamato **object file**; questo file contiene il codice macchina del programma appena tradotto, piu' una serie di importanti informazioni destinate al **linker**.

Il **linker** o collegatore, ha il compito di generare il file in formato eseguibile (estensione **.EXE** o **.COM**) a partire dal codice oggetto. Un programma **Assembly** puo' essere costituito da un unico file o da piu' file; ad esempio, se il codice sorgente si trova nei tre file chiamati **MAIN.ASM**, **LIB1.ASM** e **LIB2.ASM**, attraverso l'assembler si ottengono i tre file **MAIN.OBJ**, **LIB1.OBJ** e **LIB2.OBJ**. Il **linker** parte da questi tre moduli, e dopo aver risolto eventuali riferimenti incrociati tra essi, li collega assieme generando (se non vengono trovati errori) un file in formato eseguibile chiamato **MAIN.EXE**.

1.4 Assembler disponibili

Con il termine **assembler** generalmente si indica l'insieme formato almeno dall'assemblatore e dal linker; molti assembler forniscono spesso diversi altri strumenti molto utili, come gli editor per i file binari e i **debuggers** per la ricerca degli errori nei programmi.
Sul mercato si trovano diversi assembler, ma indubbiamente i migliori sono il **Microsoft Macro Assembler** (o **MASM**) e il **Borland Turbo Assembler** (o **TASM**). Il **MASM** e' considerato da tempo uno standard indiscusso e di conseguenza gli altri assembler offrono la possibilita' di emularlo in modo piu' o meno completo; inoltre, con una iniziativa che fara' piacere a molti, la **Microsoft** ha deciso di distribuirlo liberamente su **Internet** (si consiglia di leggere, a tale proposito, la licenza d'uso **EULA** che accompagna il **MASM**). Il **TASM**, invece, e' a pagamento ma e' decisamente piu' affidabile ed efficiente del **MASM**, tanto da guadagnarsi il titolo di assembler piu' usato ed apprezzato nel mondo **DOS/Windows**; il **MASM** ha sempre presentato diversi problemi di affidabilita' che sono stati quasi completamente risolti solo nelle versioni piu' recenti.

Nella sezione **Downloads** di questo sito e' possibile trovare sia **MASM16** (consigliato per la sezione **Assembly Base**), sia **MASM32** necessario per sviluppare programmi completi per **Windows**; se si vuole utilizzare **MASM32** per sviluppare applicazioni **DOS**, e' necessario servirsi di un apposito linker in quanto quello di **MASM32** e' riservato solo all'ambiente **Windows**. A tale proposito, dalla sezione **Downloads** e' possibile scaricare il file autoscompattante **Lnk563.exe** che contiene il linker **Microsoft** per il **DOS**; in ogni caso, in tutte le sezioni di questo sito, verra' sempre fatto riferimento ai due assembler piu' diffusi, e cioe' **MASM** e **TASM**.

Capitolo 2 - Introduzione

L'idea fondamentale che ha portato alla nascita del computer e' legata al fatto che ogni volta che si deve affrontare un qualsiasi problema, non solo matematico, si possono distinguere tre fasi ben precise:

*** la prima fase consiste nell'impostazione del problema e dei relativi dati iniziali, ed e' quella fase che possiamo definire creativa in quanto richiede l'uso del nostro cervello;**
*** la seconda fase consiste nella elaborazione dei dati che abbiamo impostato;**
*** la terza ed ultima fase consiste nella valutazione dei risultati ottenuti.**

Come si puo' intuire, la seconda fase e' quella piu' fastidiosa in quanto, una volta impostato il problema, l'elaborazione dei dati si riduce ad una sequenza di azioni da svolgere in modo meccanico e anche ripetitivo; per non parlare poi del fatto che spesso questa fase puo' richiedere tempi lunghissimi con il conseguente aumento esponenziale del rischio di commettere banali errori dovuti a qualche svista. Nel mondo dell'informatica, e' diventato famoso il caso dell'astronomo francese **Charles Delaunay**, che verso la meta' del **XVIII** secolo comincio' ad impostare un calcolo algebrico che gli avrebbe consentito di determinare con precisione la posizione della Luna rispetto alla Terra in funzione del tempo; l'impostazione di questo problema e' alla portata di qualunque astrofisico, ma cio' che scoraggerebbe chiunque, e' che lo svolgimento manuale dei calcoli richiede un tempo lunghissimo. **Delaunay** impiego' **10** anni per terminare i calcoli piu' altri **10** anni per la verifica, dopo di che pubblico' i risultati in due grossi volumi.
All'inizio degli anni **70**, gli studi di **Delaunay** cominciarono ad assumere una importanza enorme visto che si stava entrando nell'era dei satelliti artificiali, e come si sa', la Luna si comporta esattamente come un satellite artificiale in orbita attorno alla Terra. Alcuni scienziati della **Boeing Aerospace** decisero allora di sottoporre a verifica i calcoli di **Delaunay**, e a tale proposito scrissero un software capace di risolvere equazioni algebriche; con l'ausilio di questo software installato su un computer dell'epoca, il lavoro di elaborazione e verifica dei calcoli di **Delaunay** venne eseguito in circa **20** ore, con la scoperta anche di un errore dovuto ad una banale svista!
Questo esempio ci fa' capire quanto sia importante trovare il modo di delegare la fase di elaborazione dei dati di un problema, ad una macchina capace di svolgere i calcoli con estrema precisione e in tempi ridottissimi; ed e' proprio questo ragionamento che ha spinto gli esseri umani ad inventare macchine di calcolo automatico sempre piu' sofisticate ed evolute. Il prodotto finale di questa evoluzione e' rappresentato proprio dal computer.

2.1 Il concetto di algoritmo

In generale possiamo dire che il computer puo' risolvere un qualunque problema che possa essere rappresentato da una sequenza precisa di azioni da compiere; questa sequenza precisa di azioni viene definita **algoritmo**. Semplificando al massimo possiamo definire l'algoritmo come una sequenza di azioni aventi le seguenti caratteristiche:

*** queste azioni sono in numero finito;**
*** ogni azione puo' essere svolta in un tempo finito (tempo ragionevole);**
*** ogni azione successiva alla prima, e' una diretta conseguenza di quella precedente;**
*** l'algoritmo deve essere interpretabile in modo univoco da chiunque lo applichi;**
*** la sequenza di azioni deve portare sicuramente ad almeno un risultato.**

Quando si parla di algoritmi, non bisogna pensare necessariamente a problemi di tipo matematico; anche la ricetta per cuocere la pasta infatti e' un algoritmo costituito dalle seguenti azioni:

*** riempire una pentola d'acqua e metterla sul fornello acceso;**
*** versare il sale;**
*** attendere che l'acqua inizi a bollire;**
*** versare la pasta;**
*** attendere per il tempo di cottura specificato nella confezione;**
*** scolare la pasta;**
*** condire la pasta;**
*** servire in tavola.**

Le azioni da compiere per cuocere la pasta sono in numero finito e ciascuna di esse viene svolta in un tempo ragionevole nel senso che ad esempio il tempo di cottura non richiede certo **20** anni; inoltre la sequenza delle azioni e' chiara ed interpretabile da tutti in modo univoco visto che ogni azione successiva alla prima e' un'ovvia conseguenza dell'azione precedente.

2.2 L'architettura di Von Neumann

Riassumendo quindi, possiamo dire che la risoluzione di un problema con l'ausilio del computer, puo' essere separata in tre fasi:

*** impostazione dei dati del problema;**
*** elaborazione dei dati;**
*** presentazione dei risultati.**

Tutte le macchine di calcolo automatico posseggono in linea di principio, una struttura legata allo schema appena presentato; questa struttura viene definita **architettura di Von Neumann** dal nome dello scienziato che l'ha teorizzata. Una macchina basata sull'architettura di **Von Neumann** e' quindi costituita da tre blocchi principali:

*** un dispositivo per l'inserimento dei dati (input);**
*** un dispositivo per l'elaborazione dei dati (unita' di elaborazione);**
*** un dispositivo per la presentazione dei risultati (output).**

Consideriamo ad esempio un generico computer caratterizzato da una struttura estremamente semplificata; in questo caso possiamo individuare la **tastiera** come dispositivo di **input** dei dati. Attraverso la tastiera e un editor di testo possiamo scrivere (cioe' inserire) un programma contenente sia i dati in **input**, sia l'algoritmo che dovra' elaborare questi dati; queste informazioni vengono usualmente immagazzinate in un dispositivo chiamato **memoria**. Come dispositivo per l'elaborazione dei dati possiamo individuare il **microprocessore** (o **CPU**); infine, come dispositivo di **output** possiamo individuare lo **schermo** del computer.

2.3 Codifica dell'informazione

Una volta stabilito che vogliamo usare il computer per risolvere un problema, rimane da superare un ostacolo che ci appare abbastanza impegnativo: come puo' una macchina come il computer capire cosa sia un numero, un colore, una lettera dell'alfabeto, un segno di punteggiatura, una figura geometrica, una equazione algebrica?
Per superare questo ostacolo, la cosa migliore da fare consiste nel trovare un sistema che ci permetta di rappresentare queste entita' attraverso un apposito codice comprensibile dal computer; appare intuitivo il fatto che il sistema di codifica scelto deve anche essere il piu' semplice possibile perche' questo si traduce poi in una conseguente semplificazione dell'architettura della macchina (semplificazione circuitale nel caso del computer). Non bisogna fare molti sforzi per capire che il

sistema di codifica piu' semplice consiste nell'utilizzare i numeri, attraverso i quali possiamo rappresentare teoricamente qualunque cosa; vediamo allora quale puo' essere il sistema di numerazione piu' adatto per la codifica dell'informazione sul computer.

Nel corso della storia gli esseri umani hanno concepito diversi sistemi di numerazione che possiamo raggruppare in due categorie: la prima comprende i sistemi di numerazione basati su precise regole logico matematiche, mentre la seconda comprende i sistemi di numerazione basati invece solo su aspetti di carattere simbolico; questa seconda categoria naturalmente e' da escludere non avendo nessun fondamento logico. Si puo' citare ad esempio il sistema di numerazione simbolico usato dai romani che inizialmente veniva utilizzato per rappresentare numeri molto piccoli (come ad esempio gli anni); ogni volta che si presentava la necessita' di rappresentare un numero piu' grande del solito, si introduceva arbitrariamente un nuovo simbolo.
Nel sistema di numerazione romano il simbolo **I** rappresenta una unita', il simbolo **II** rappresenta due unita' (una piu' una), mentre il simbolo **III** rappresenta tre unita' (una piu' una piu' una); a questo punto, seguendo la logica il simbolo **IIII** dovrebbe rappresentare quattro unita'. I romani pero' come sappiamo hanno introdotto il nuovo simbolo **V** per indicare cinque unita', e hanno rappresentato quattro unita' con il simbolo **IV** (cinque meno una); in sostanza, se la barretta appare alla destra del simbolo **V** viene sommata, mentre se appare alla sinistra viene sottratta. Seguendo sempre la logica, per rappresentare dieci unita' dovremmo allora scrivere **VV** (cinque piu' cinque); anche in questo caso pero' i romani hanno introdotto il nuovo simbolo **X** per rappresentare dieci unita'. Questa situazione poco razionale si ripete con il simbolo **L** che rappresenta cinquanta unita', con il simbolo **C** che rappresenta cento unita' e cosi' via; appare evidente il fatto che non avrebbe nessun senso pensare di implementare sul computer un sistema di questo genere. Tra i principali difetti del sistema di numerazione romano si puo' citare ad esempio l'ambiguita' di rappresentazione (ad esempio, per rappresentare otto unita' si potrebbe utilizzare il simbolo **VIII**, oppure il simbolo **IIX**); un'altro difetto e' dato dall'assenza di un simbolo per la rappresentazione dello zero.

Non ci resta allora che ricorrere ai sistemi di numerazione basati su precise regole logico matematiche; tra questi sistemi di numerazione, il piu' razionale appare quello anticamente in uso presso le popolazioni Indu', e trasmesso poi attraverso gli arabi anche nell'Europa occidentale nel periodo medioevale (per questo motivo si parla anche di sistema di numerazione arabo). Questo sistema di numerazione consiste nel definire un insieme **S** di simboli chiamati **cifre**; una sequenza di cifre rappresenta un **numero**. Il numero dei simboli usati prende il nome di **base del sistema di numerazione** (**b**); nel mondo occidentale viene ad esempio utilizzato un sistema di numerazione costituito come sappiamo dal seguente insieme di dieci simboli:
`S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`
Proprio per questo motivo si parla di **sistema di numerazione in base 10** (**b = 10**).
Il sistema arabo viene chiamato anche **posizionale** e questa e' sicuramente una delle caratteristiche piu' importanti; posizionale significa che le cifre che compongono il numero assumono un diverso **peso** a seconda della loro posizione nel numero stesso. Nel caso del sistema arabo, il peso delle cifre cresce scorrendo il numero da destra verso sinistra; si ricordi a tale proposito che gli arabi scrivono da destra verso sinistra.
Consideriamo ad esempio il numero **14853** espresso nel sistema di numerazione posizionale in base **10**; possiamo dire che la cifra **3** (cifra meno significativa) ha posizione **p=0**, la cifra **5** ha posizione **p=1**, la cifra **8** ha posizione **p=2**, la cifra **4** ha posizione **p=3** e infine la cifra **1** (cifra piu' significativa) ha posizione **p=4**.
Per capire bene questo concetto importantissimo, si osservi che i **10** simboli usati rappresentano le cosiddette **unita'** (zero unita', una unita', due unita', ... , nove unita'); aggiungendo una unita' a nove unita', non sappiamo come rappresentare questo numero perche' non ci sono piu' simboli disponibili. Invece di inventare un nuovo simbolo (come nel sistema di numerazione dei romani), passiamo alle **decine**, cioe' ai gruppi di dieci unita' rappresentando il numero:

9 + 1 come 10, cioe' zero unita' piu' una decina; con le decine possiamo arrivare sino a **99** (nove unita' piu' nove decine). Aggiungendo una unita' a **99** unita', ci imbattiamo nella situazione precedente; introduciamo allora le **centinaia**, cioe' i gruppi di cento unita' rappresentando il numero:
99 + 1 come 100, cioe' zero unita' piu' zero decine piu' una centinaia. A questo punto il meccanismo e' chiaro e si intuisce subito che con questo sistema si possono rappresentare infiniti numeri.
Vediamo un esempio pratico rappresentato dal numero **333**; il numero **333** e' formato da tre cifre uguali ma il loro peso e' differente. Osserviamo infatti che il **3** di destra rappresenta tre unita', il **3** centrale rappresenta tre decine, mentre il **3** di sinistra rappresenta tre centinaia; possiamo scrivere allora:

333 = 300 + 30 + 3 = (3 * 100) + (3 * 10) + (3 * 1)

Tenendo presente che:

$1 = 10^0$
$10 = 10^1$
$100 = 10^2$

possiamo scrivere:

333 = 300 + 30 + 3 =
(3 * 100) + (3 * 10) + (3 * 1) =
$(3 * 10^2) + (3 * 10^1) + (3 * 10^0)$

Dato un numero **n** in base **b** formato da **i** cifre aventi posizione:
p = 0, 1, 2, ... i - 1 (a partire dalla cifra piu' a destra), si definisce peso di una cifra di **n** la potenza di ordine **p** della base **b**; dato allora il numero **333**:

***** il peso del **3** piu' a destra e' $\mathbf{10^0 = 1}$;
***** il peso del **3** centrale e' $\mathbf{10^1 = 10}$;
***** il peso del **3** piu' a sinistra e' $\mathbf{10^2 = 100}$.

Con questo sistema e' facile rappresentare anche i numeri con la virgola; ad esempio:

123.456 = 100 + 20 + 3 + 0.4 + 0.05 + 0.006 =
(1 * 100) + (2 * 10) + (3 * 1) + (4 * 0.1) + (5 * 0.01) + (6 * 0.001) =
$(1 * 10^2) + (2 * 10^1) + (3 * 10^0) + (4 * 10^{-1}) + (5 * 10^{-2}) + (6 * 10^{-3})$

La posizione delle cifre (da destra a sinistra) e':

-3, -2, -1, 0, 1, 2

Il peso delle cifre (da destra a sinistra) e':

$10^{-3}, 10^{-2}, 10^{-1}, 10^0, 10^1, 10^2$

Possiamo dire in definitiva (anche per quanto si vedra' in seguito) che il sistema posizionale arabo appare il piu' adatto per la codifica dell'informazione in una macchina di calcolo automatico.

Abbiamo gia' incontrato un primo esempio di applicazione di questo sistema di codifica tramite i numeri, e cioe' il set di codici ASCII che definisce un insieme di **256** simboli riconosciuti dai **PC** di tutto il mondo; si tratta di un vero e proprio alfabeto del computer costituito da codici di controllo, lettere maiuscole e minuscole dell'alfabeto anglosassone, cifre, segni di punteggiatura, etc. Questi

256 simboli vengono rappresentati da **256** codici numerici (da **0** a **255**) che permettono a qualunque **PC** di interpretare allo stesso modo una determinata sequenza di codici; ad esempio la sequenza:

```
65 115 115 101 109 98 108 121
```

verra' visualizzata da un editor di testo come la stringa:

```
Assembly
```

su qualunque **PC** e in qualunque parte del mondo. Naturalmente, con la diffusione planetaria dei computers e con l'interconnessione mondiale tra computers grazie ad **Internet**, **256** simboli appaiono assolutamente insufficienti (si pensi solo all'enorme numero di simboli linguistici usati in Cina, Giappone, Russia, India etc.); per questo motivo si sta diffondendo internazionalmente tra i computers il nuovo sistema di codifica **UNICODE** che puo' contenere sino a **65536** simboli. I primi **128** simboli dell'**UNICODE** coincidono esattamente con i primi **128** simboli del codice ASCII. Per maggiori informazioni sull'**UNICODE** si puo' visitare il sito ufficiale:

```
http://www.unicode.org
```

Oltre ai codici ASCII e **UNICODE**, si possono fare numerosi altri esempi di codifica numerica dell'informazione sul computer; considerando ad esempio la tastiera, ad ogni tasto viene associato un codice numerico chiamato **codice di scansione**. Questi codici di scansione o Scan Codes, consentono al computer di sapere quale tasto e' stato premuto; generalmente, i programmi di gestione della tastiera leggono il codice di scansione del tasto appena premuto, e lo convertono nel codice ASCII del simbolo stampato sul tasto stesso.
Un'altro esempio e' legato al mouse il cui programma di gestione determina istante per istante le coordinate **x**, **y** (ascissa e ordinata) della posizione del cursore sullo schermo; ad ogni pulsante del mouse inoltre viene associato un apposito codice numerico che ci permette di sapere se il pulsante stesso e' premuto o meno.
Un ulteriore esempio riguarda la gestione dello schermo grafico del computer; supponendo ad esempio che una schermata grafica sia costituita da una matrice di **320 x 200** punti (**pixel**), e che ciascun punto possa assumere uno tra **256** colori distinti, possiamo rappresentare tutti i punti dello schermo attraverso un vettore di **320 x 200 = 64000** elementi dove ogni elemento e' un numero compreso tra **0** e **255** che rappresenta il colore del punto stesso. Per accedere in lettura o in scrittura ad un pixel dello schermo, basta specificare l'indice (cioe' un numero compreso tra **0** e **63999**) del vettore dei pixel; a questo punto e' possibile leggere il colore di quel pixel o modificarlo.

Tutte le considerazioni appena svolte ci fanno capire che per impostare sul computer un determinato problema, dobbiamo prima convertire il problema stesso in una serie di codici numerici, cioe' in un formato comprensibile dal computer; questa fase prende anche il nome di **codifica dell'informazione**. A questo punto si passa alla fase di elaborazione che viene svolta dalla **CPU** del computer; e' facile convincersi che questa fase consiste nell'applicazione di una serie di operazioni matematiche sui codici numerici appena inseriti. Per rendercene conto, vediamo un esempio pratico molto semplice.
Supponiamo di avere una sequenza di lettere dell'alfabeto, cioe' una **stringa di testo**, che vogliamo convertire nella stessa stringa scritta pero' tutta in maiuscolo; e' chiaro che la stringa per poter essere gestita dal computer, deve essere espressa sotto forma di codici numerici. Come abbiamo visto in precedenza, la codifica di una stringa puo' essere effettuata attraverso il codice ASCII; in sostanza possiamo dire che una stringa di testo viene gestita dal computer sotto forma di vettore di codici ASCII. Una volta che la stringa e' stata convertita in codici numerici, la fase di elaborazione consiste in una sequenza di istruzioni che esaminano il codice numerico di ogni carattere, e se questo codice appartiene ad un carattere minuscolo, lo convertono nel codice del corrispondente carattere maiuscolo; l'insieme di tutte queste istruzioni costituisce il **programma di elaborazione**, e cioe' l'algoritmo che effettua la conversione da minuscolo in maiuscolo.
La fase di output consiste nel mostrare sullo schermo questa nuova sequenza di lettere convertite in maiuscolo; in questo modo l'utente puo' vedere sullo schermo il risultato delle elaborazioni.
Il programma che elabora la stringa e' molto semplice; osservando infatti la tabella dei codici

ASCII, possiamo notare che le **26** lettere minuscole dell'alfabeto inglese, hanno codici numerici che vanno da **97** a **122**, e inoltre questi codici sono consecutivi e contigui. Le **26** lettere maiuscole hanno invece codici numerici che vanno da **65** a **90**, e anche in questo caso questi codici sono consecutivi e contigui; di conseguenza, tra il codice di una lettera minuscola e il codice della stessa lettera maiuscola esiste una distanza fissa pari a **32**. Nel caso ad esempio della lettera '**a**' abbiamo:

```
ASCII('A') = 65
ASCII('a') = 97
ASCII('a') - ASCII('A') = 97 - 65 = 32
```

E' chiaro quindi che per convertire una lettera minuscola in maiuscola, basta sottrarre **32** al suo codice numerico; il programma di elaborazione (scritto in pseudo codice) e' costituito allora dalle seguenti istruzioni:

```
1) inizia dal primo carattere della stringa
2) se il codice ASCII e' compreso tra 65 e 90 salta al punto 4
3) sottrai 32 al codice corrente
4) passa al carattere successivo
5) se la stringa non e' terminata salta al punto 2
```

Come si puo' notare da questo esempio, tutto sul computer e' rappresentato da numeri; le stringhe di testo non sono altro che una sequenza di codici ASCII (cioe' una sequenza di numeri). Per sapere se un carattere della stringa e' minuscolo, bisogna verificare se il suo codice numerico e' compreso tra **97** e **112** (confronto tra numeri); per convertire un carattere minuscolo in maiuscolo bisogna sottrarre al suo codice numerico il valore **32** (sottrazione tra numeri).

Visto e considerato che qualunque informazione viene gestita dal computer attraverso dei codici numerici, l'elaborazione di queste informazioni consiste nell'eseguire su questi codici numerici delle operazioni matematiche; la conseguenza di tutto cio' e' che per capire il modo di lavorare del computer, bisogna studiare in dettaglio quella che possiamo definire la **matematica del computer**. Questo e' proprio l'argomento del prossimo capitolo.

Capitolo 3 - La matematica del computer

Il concetto fondamentale emerso nel precedente capitolo, e' che il computer gestisce qualunque tipo di informazione sotto forma di codici numerici; di conseguenza, elaborare queste informazioni significa elaborare numeri, cioe' eseguire varie operazioni matematiche su questi numeri. Lo scopo di questo capitolo e' proprio quello di analizzare in dettaglio tutti gli aspetti relativi a quella che puo' essere definita la **matematica del computer**.
Abbiamo visto che la rappresentazione numerica piu' adatta per una macchina come il computer e' il sistema posizionale arabo e abbiamo anche visto che con questo sistema non esistono limiti teorici alla dimensione del numero che si vuole rappresentare; il problema e' che mentre noi esseri umani abbiamo la capacita' di immaginare numeri enormi, il computer non puo' operare su numeri frutto dell'immaginazione ma li deve gestire fisicamente in apposite aree interne che possiamo paragonare a dei contenitori. Le dimensioni di questi contenitori sono limitate e di conseguenza anche la massima dimensione dei numeri che devono contenere sara' limitata.
Il problema che si presenta sul computer viene efficacemente rappresentato dall'esempio del contachilometri delle automobili. Supponiamo di comprare un'auto nuova dotata di contachilometri a **5** cifre che inizialmente segna **00000**; durante i viaggi il contachilometri si incrementa di una unita' ad ogni chilometro sino ad arrivare a **99999**. A questo punto, se percorriamo un'altro chilometro, il contachilometri avendo solo **5** cifre non segnera' **100000** ma **00000**!
Questo e' esattamente cio' che accade sul computer quando ad esempio si sommano due grossi numeri ottenendo un risultato che eccede la dimensione massima consentita; in questo caso il computer ci dara' un risultato imprevisto. Questo problema sembra estremamente grave e tale da rendere i computers (in assenza di soluzioni) assolutamente inutilizzabili; come pero' vedremo in seguito, per fortuna le soluzioni esistono. E' importante sottolineare che l'esempio del contachilometri ha una importanza straordinaria in quanto ci consentira' anche in seguito di capire perfettamente il modo di operare del computer; per studiare la matematica del computer ci serviremo inizialmente di un computer "immaginario" che conta in base **10**.

3.1 Un computer immaginario che lavora in base 10

Cominciamo allora con lo stabilire che il nostro computer "immaginario" lavori con il sistema di numerazione posizionale arabo in base **b=10**, e sia in grado di gestire numeri a **5** cifre; con **5** cifre possiamo rappresentare tutti i numeri interi compresi tra **0** e **99999**, per un totale di **100000** numeri differenti ($\mathbf{10^5}$). A proposito di numero di cifre, spesso si sente parlare di computers con architettura a **16** bit, **32** bit, **64** bit etc; questi numeri rappresentano in pratica il numero massimo di cifre che l'hardware del computer puo' manipolare in un colpo solo. Possiamo dire allora che il nostro computer immaginario avra' una architettura a **5** cifre per la rappresentazione di numeri in base **10**; i **100000** numeri ottenibili con queste **5** cifre verranno utilizzati per la codifica numerica delle informazioni da gestire attraverso il computer.
Un'altro aspetto importante da chiarire e': che tipo di operazioni matematiche puo' eseguire il computer su questi numeri?
Sarebbe piuttosto complicato se non impossibile realizzare un computer capace di calcolare direttamente funzioni trigonometriche, logaritmiche, esponenziali, etc; per fortuna la matematica ci viene incontro dimostrandoci che e' possibile approssimare in modo piu' o meno semplice qualunque (o quasi) funzione con un polinomio di grado prestabilito contenente quindi solamente le quattro operazioni fondamentali e cioe': addizione, sottrazione, moltiplicazione e divisione. Questo aspetto e' di grandissima importanza per il computer in quanto consente di ottenere enormi semplificazioni circuitali nel momento in cui si devono implementare via hardware questi operatori matematici; in seguito vedremo addirittura che il computer in realta' e' in grado svolgere il proprio lavoro servendosi esclusivamente di addizioni, scorrimenti di cifre, cambiamenti di segno e pochi altri semplicissimi operatori.

A questo punto ci interessa vedere come si applicano le quattro operazioni fondamentali ai codici a **5** cifre del nostro computer ricordando che i circuiti elettronici preposti ad eseguire le operazioni si trovano come sappiamo nella **CPU**.

3.2 Numeri interi senza segno

In matematica l'insieme numerico rappresentato dalla sequenza:
`1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...`
viene chiamato **insieme dei numeri naturali**, e viene indicato con il simbolo **N**; si tratta in sostanza dell'insieme dei numeri interi strettamente positivi per i quali e' implicito il segno **+** davanti al numero stesso (per questo motivo si parla anche di numeri interi senza segno). All'insieme **N** viene in genere aggiunto anche lo zero che convenzionalmente e' privo di segno in quanto rappresenta una quantita' nulla; da questo momento in poi indicheremo con **N** l'insieme dei numeri naturali piu' lo zero. Vediamo allora come bisogna procedere per codificare l'insieme **N** sul nostro computer; e' chiaro che la **CPU** sara' in grado di rappresentare solo un sottoinsieme finito degli infiniti numeri appartenenti ad **N**.
L'implementazione dell'insieme **N** e' piuttosto semplice ed intuitiva; il nostro computer infatti ha una architettura a **5** cifre, e con **5** cifre possiamo codificare tutti i numeri interi senza segno compresi tra **0** e **99999**. Si ottiene quindi la codifica rappresentata in Figura 1:

Figura 1
```
00000d rappresenta il valore      0
00001d rappresenta il valore     +1
00002d rappresenta il valore     +2
00003d rappresenta il valore     +3
..................................
99996d rappresenta il valore +99996
99997d rappresenta il valore +99997
99998d rappresenta il valore +99998
99999d rappresenta il valore +99999
```

In Figura 1 tutti i numeri che terminano con una **d** rappresentano un codice numerico del computer; nel nostro caso si tratta di codici numerici espressi da numeri interi in base **10**, formati da **5** cifre ciascuno. In questo modo possiamo evitare di fare confusione tra una determinata informazione e la corrispondente codifica numerica usata dal computer; da questo momento in poi verra' sempre utilizzato questo accorgimento.
La Figura 1 ci mostra quindi che il computer rappresenta il numero **0** attraverso il codice **00000d**, il numero **1** attraverso il codice **00001d**, il numero **2** attraverso il codice **00002d** e cosi' via; attraverso questo sistema di codifica, la nostra **CPU** puo' gestire via hardware un sottoinsieme finito di **N** formato quindi dai seguenti **100000** numeri naturali:
`0, 1, 2, 3, ..., 99996, 99997, 99998, 99999`

3.2.1 Addizione tra numeri interi senza segno

Con il sistema di codifica appena illustrato, le addizioni non ci creano problemi a meno che la somma dei due addendi non superi **99999**, cioe' il numero piu' grande che il computer puo' rappresentare con **5** cifre; la Figura 2 illustra i vari casi che si possono presentare:

Figura 2
```
1) 44000d + 55000d = 99000d                         CF = 0
2) 37241d + 28999d = 66240d                         CF = 0
3) 50000d + 60000d = 10000d (invece di 110000d)  CF = 1
4) 99999d + 00001d = 00000d (invece di 100000d)  CF = 1
```

Come si puo' notare, gli esempi **1** e **2** forniscono il risultato che ci aspettavamo, mentre i risultati degli esempi **3** e **4** sono (apparentemente) privi di senso; ricordando l'esempio del contachilometri, siamo in grado di capire cosa e' successo. La **CPU** ha eseguito i calcoli correttamente, ma ha dovuto troncare la cifra piu' significativa (cioe' quella piu' a sinistra che e' anche la cifra di peso maggiore) perche' non e' in grado di rappresentare un numero di **6** cifre; ecco quindi che **110000d** diventa **10000d** e **100000d** diventa **00000d**.
Quello che sembra un grave problema, viene risolto in modo piuttosto semplice attraverso un sistema che consente alla **CPU** di dare al programmatore la possibilita' di interpretare correttamente il risultato; ogni **CPU** e' dotata di una serie di "segnalatori" chiamati **flags** attraverso i quali e' possibile avere un controllo totale sul risultato delle operazioni appena eseguite.
Il primo flag che incontriamo si chiama **Carry Flag** (**CF**) e cioe' **segnalatore di riporto**; nella Figura 2 e' riportato a fianco di ogni risultato lo stato di **CF** che vale **0** se il risultato non produce un riporto, vale invece **1** se si e' verificato un riporto. Ecco quindi che nei primi due casi **CF=0** indica che il risultato e' valido cosi' com'e'; nel terzo caso **CF=1** indica che il risultato deve essere interpretato come **10000** con riporto di **1** (cioe' con riporto di un centinaio di migliaia). Nel quarto caso **CF=1** indica che il risultato deve essere interpretato come **00000** con riporto di **1** (cioe' con riporto di un centinaio di migliaia).
In sostanza il programmatore subito dopo aver effettuato una addizione, deve verificare lo stato di **CF**; in questo modo ha la possibilita' di sapere se si e' verificato o meno un riporto. A questo punto si intuisce subito il fatto che grazie a **CF** possiamo addirittura effettuare somme tra numeri teoricamente di qualunque dimensione superando cosi' le limitazioni imposteci dall'hardware; la tecnica da utilizzare e' la stessa che ci viene insegnata meccanicamente alle scuole elementari, e che adesso possiamo chiarire meglio sfruttando le conoscenze che abbiamo acquisito nel precedente capitolo. Supponiamo ad esempio di voler eseguire la seguente somma:

```
375 +
249 =
-----
```

Perche' il numero viene incolonnato in questo modo?
La risposta e' data dal fatto che stiamo usando il sistema posizionale arabo, e quindi per eseguire correttamente una somma dobbiamo incolonnare unita' con unita', decine con decine, centinaia con centinaia, etc. Partendo allora dalla colonna delle unita' si ottiene:
`5 unita' + 9 unita' = 14 unita'`
cioe' **4** unita' con riporto di una decina che verra' aggiunta alla colonna delle decine.
Passando alla colonna delle decine otteniamo:
`7 decine + 4 decine = 11 decine`
piu' una decina di riporto = **12** decine pari a **120** unita', cioe' **2** decine con riporto di un centinaio che verra' aggiunto alla colonna delle centinaia.
Passando infine alla colonna delle centinaia otteniamo:
`3 centinaia + 2 centinaia = 5 centinaia`
piu' un centinaio di riporto = **6** centinaia; il risultato finale sara' quindi **624**.
Si puo' subito notare che sommando una colonna qualsiasi, l'eventuale riporto non sara' mai superiore a **1**; infatti il caso peggiore per la prima colonna e':
`9 + 9 = 18`
cioe' **8** con riporto di **1**. Per le colonne successive alla prima, anche in presenza del riporto (**1**) il caso peggiore e':
`9 + 9 + 1 = 19`
cioe' **9** con riporto di **1**.
Simuliamo ora questo procedimento sul nostro computer sfruttando pero' il fatto che la nostra **CPU** puo' manipolare direttamente (via hardware) numeri a **5** cifre; questo significa che per sommare numeri con piu' di **5** cifre, basta spezzarli in gruppi di **5** cifre (partendo da destra). La **CPU** somma

via hardware due gruppi corrispondenti di **5** cifre fornendoci alla fine l'eventuale riporto attraverso **CF**; il contenuto di **CF** puo' essere utilizzato quindi per la somma successiva. Anche in questo caso, sommando tra loro gruppi di **5** cifre, l'eventuale riporto finale non potra' mai superare **1**; osserviamo infatti che la somma massima e':

```
99999 + 99999 = 199998
```

cioe' **99998** con riporto di **1**. Se c'era gia' un riporto si ottiene **199999**, cioe' **99999** con riporto di **1**; in definitiva, possiamo dire che **CF** contiene il riporto (**0** o **1**) della somma appena effettuata.
Per effettuare quindi con la nostra **CPU** la somma tra due numeri, sara' necessario incolonnare non le cifre corrispondenti, ma i gruppi di **5** cifre corrispondenti; in sostanza, le colonne dell'addizione sono formate non da cifre ma da gruppi di **5** cifre. Partendo dalla colonna piu' a destra facciamo eseguire la prima somma alla **CPU**; in seguito passiamo alla colonna successiva ed eseguiamo la nuova somma tenendo conto dell'eventuale riporto prodotto dalla somma precedente.
Per chiarire meglio questi concetti, vediamo in Figura 3 un esempio relativo a numeri che possono avere sino a **15** cifre:

Figura 3

```
Vogliamo calcolare: 3216679288700 + 4425532140000 ( = 7642211428700)

Spezziamo i due numeri in gruppi di 5 cifre ottenendo:

00321d 66792d 88700d +
00442d 55321d 40000d =
-------------------

Incolonniamo i gruppi corrispondenti di 5 cifre
ed eseguiamo le varie somme colonna per colonna:

colonna 1: 88700d + 40000d = 28700d      (CF = 1)
colonna 2: 66792d + 55321d + CF = 22114d (CF = 1)
colonna 3: 00321d + 00442d + CF = 00764d (CF = 0)

Unendo i tre risultati si ottiene: 00764 22114 28700
```

La terza ed ultima somma ci da' **CF=0**, il che significa che non c'e' un ulteriore riporto e quindi il risultato finale e' valido cosi' com'e'; se alla fine avessimo ottenuto **CF=1**, avremmo dovuto aggiungere un **1** alla sinistra delle **15** cifre del risultato.
Con questo sistema, e' possibile sommare numeri con (in teoria) un qualunque numero di cifre, suddividendoli in gruppi di **5** cifre che possono essere gestiti direttamente dalla **CPU**; naturalmente la gestione degli eventuali riporti e' a carico del programmatore che dovra' tenerne conto nel programma che esegue la somma.
E' chiaro che se la **CPU** fosse in grado di manipolare direttamente (via hardware) numeri di **15** cifre, eseguirebbe le somme alla massima velocita' possibile; nel nostro caso invece siamo costretti a seguire il metodo appena visto (via software) con conseguente diminuzione delle prestazioni. Le industrie che producono hardware cercano di realizzare **CPU** con architetture sempre piu' ampie proprio perche' questo significa aumentare in modo notevole le prestazioni.

3.2.2 Sottrazione tra numeri interi senza segno

Passando alle sottrazioni, sappiamo che nell'insieme **N** questa operazione e' possibile solo quando il primo numero (**minuendo**) e' maggiore o almeno uguale al secondo numero (**sottraendo**); se invece il minuendo e' minore del sottraendo, il risultato che si ottiene e' negativo e quindi non appartiene ad **N**. In Figura 4 vediamo come si comporta la **CPU** nei vari casi possibili:

Figura 4

```
1) 32000d - 21000d = 11000d                        CF = 0
2) 37241d - 37241d = 00000d                        CF = 0
3) 50000d - 60000d = 90000d (invece di -10000)  CF = 1
4) 00100d - 50000d = 50100d (invece di -49900)  CF = 1
```

Nei primi due esempi di Figura 4 la **CPU** restituisce il risultato che ci aspettavamo in quanto il minuendo e' maggiore (primo esempio) o almeno uguale (secondo esempio) al sottraendo; negli esempi **3** e **4** invece (minuendo minore del sottraendo), i risultati forniti appaiono piuttosto strani. Notiamo pero' che anche nel caso della sottrazione, la **CPU** modifica **CF** che viene posto a **1** ogni volta che si dovrebbe ottenere un risultato negativo; come deve essere interpretata questa situazione?
Questa volta **CF** non deve essere interpretato come la segnalazione di un riporto, ma trattandosi di una sottrazione, **CF** sta segnalando un prestito (**borrow**); la **CPU** cioe' si comporta come se il minuendo continuasse verso sinistra con altri gruppi di **5** cifre dai quali chiedere il prestito. A questo punto possiamo spiegarci il risultato degli esempi **3** e **4** della Figura 3:
`3) 50000 - 60000 = 90000`
con prestito di **1** (cioe' di **100000**) dall'ipotetico successivo gruppo di **5** cifre del minuendo; infatti:
`(50000 + 100000) - 60000 = 150000 - 60000 = 90000`
con **CF=1** che segnala il prestito.
`4) 00100 - 50000 = 50100`
con prestito di **1** (cioe' di **100000**) dall'ipotetico successivo gruppo di **5** cifre del minuendo; infatti:
`(00100 + 100000) - 50000 = 100100 - 50000 = 50100`
con **CF=1** che segnala il prestito.
La **CPU** quindi simula il meccanismo che ci viene insegnato alle scuole elementari per calcolare una sottrazione; supponiamo ad esempio di voler calcolare:

```
300 -
009 =
-----
```

Come nel caso dell'addizione, i due numeri vengono incolonnati in questo modo in quanto dobbiamo sottrarre unita' da unita', decine da decine, centinaia da centinaia, etc.
Partendo allora dalla colonna delle unita' si vede subito che da **0** unita' non si possono sottrarre **9** unita'; si chiede allora un prestito di **10** unita' (cioe' di una decina) alle decine del minuendo, ma non essendoci decine, si chiede allora un prestito di **10** unita' alle centinaia del minuendo che diventano cosi':
`300 - 10 = 290`
Possiamo ora sottrarre la prima colonna ottenendo:
`10 - 9 = 1`
Passiamo alla seconda colonna dove troviamo le **9** decine rimaste in seguito al prestito precedente; la sottrazione relativa alla seconda colonna ci da':
`9 - 0 = 9`
Passiamo infine alla terza colonna dove troviamo le **2** centinaia rimaste, e quindi:
`2 - 0 = 2`
Il risultato finale e': **291**.

Attraverso questo sistema la **CPU** ci permette (come per l'addizione) di sottrarre tra loro numeri interi senza segno di qualunque dimensione; come al solito la tecnica da adottare consiste nel suddividere sia il minuendo che il sottraendo in gruppi di **5** cifre a partire da destra. La Figura 5 mostra un esempio pratico relativo alla sottrazione tra numeri da **8** cifre ciascuno:

Figura 5

```
Vogliamo calcolare 56127834 - 32860521 ( = 23267313)

Spezziamo i due numeri in gruppi di 5 cifre ottenendo:

00561d 27834d -
00328d 60521d =
-------------

Incolonniamo i gruppi corrispondenti di 5 cifre ed
eseguiamo le varie sottrazioni colonna per colonna:

colonna 1: 27834d - 60521d = 67313d        (CF = 1)
colonna 2: (00561d - CF) - 00328d = 00232d (CF = 0)

Unendo i due risultati si ottiene 00232 67313
```

La seconda ed ultima sottrazione ci da' **CF=0** per indicare che non c'e' stato un ulteriore prestito e quindi il risultato finale e' valido cosi' com'e'; se avessimo ottenuto **CF=1** il risultato sarebbe stato negativo e quindi non appartenente ad **N**.
Come si puo' notare, la sottrazione relativa alla prima colonna non deve tenere conto di nessun prestito precedente; le sottrazioni relative invece a tutte le colonne successive alla prima devono verificare il contenuto di **CF** per sapere se c'e' stato un prestito richiesto dalla precedente sottrazione. Anche in questo caso si puo' dimostrare che l'eventuale prestito e' rappresentato proprio dal valore **0** o **1** contenuto in **CF**; infatti, cosi' come nell'addizione il riporto massimo e' **1**, anche nella sottrazione il prestito massimo non supera mai **1**. Osserviamo a tale proposito che in relazione alla sottrazione della prima colonna, il caso peggiore che si puo' verificare e':

```
00000 - 99999
```

che richiede un prestito di **1** dalla colonna successiva; in relazione invece alle sottrazioni delle colonne successive alla prima, il caso peggiore che si puo' presentare e':

```
(00000 - CF) - 99999
```

che anche nel caso di **CF=1** richiedera' al massimo un prestito di **1** dalla colonna successiva.
In questo secondo caso si puo' notare che vengono effettuate due sottrazioni per tenere conto anche di **CF**; e' evidente pero' che queste due sottrazioni non potranno mai richiedere due prestiti. Infatti, se il minuendo vale **00000** e **CF** vale **1**, allora la prima sottrazione richiede un prestito che trasforma il minuendo in:

```
100000 - CF = 100000 - 1 = 99999
```

Di conseguenza la seconda sottrazione non potra' mai richiedere un prestito perche' il sottraendo non puo' essere maggiore di **99999**.
Se il minuendo e' maggiore di **00000** (ad esempio **00001**), allora la prima sottrazione non potra' mai richiedere un prestito in quanto **CF** vale al massimo **1**; il prestito (di **1**) potra' essere necessario eventualmente per la seconda sottrazione.

Ci si puo' chiedere come debba essere interpretato l'esempio **3** della Figura 4:

```
3) 50000d - 60000d = 90000d (invece di -10000) CF = 1
```

nel caso in cui si stia operando su numeri interi di sole **5** cifre; la risposta e' che nell'insieme **N** questo risultato non ha senso perche' in questo caso il minuendo non puo' essere minore del sottraendo. Il risultato invece ha senso nell'insieme dei numeri interi positivi e negativi (detti numeri con segno o numeri relativi); in questo caso infatti, come vedremo piu' avanti **90000d** e' proprio la

codifica numerica del numero negativo **-10000**!

Per capire ora il vero significato di **CF** analizziamo i valori assunti in sequenza da un numero a **5** cifre che viene via via incrementato di una unita' per volta (o che viene via via decrementato di una unita' per volta):
`....., 99996, 99997, 99998, 99999, 00000, 00001, 00002, 00003, .....`
In questa sequenza possiamo individuare un punto molto importante rappresentato dal "confine" che separa **99999** e **00000**; passando da **99999** a **00000** attraversiamo questo confine da sinistra a destra, mentre passando da **00000** a **99999** attraversiamo questo confine da destra a sinistra.
Osserviamo ora che nel caso dell'addizione tra numeri interi senza segno, se il risultato finale non supera **99999** (**CF=0**), vuol dire che siamo rimasti alla sinistra del confine; se invece il risultato finale supera **99999** (**CF=1**), vuol dire che abbiamo oltrepassato il confine procedendo da sinistra a destra.
Passando alle sottrazioni tra numeri interi senza segno, se il risultato finale e' maggiore o uguale a **00000** (**CF=0**), vuol dire che siamo rimasti alla destra del confine; se invece il risultato finale e' minore di **00000** (**CF=1**), vuol dire che abbiamo oltrepassato il confine procedendo da destra verso sinistra.
In definitiva possiamo dire che **CF=1** segnala il fatto che nel corso di una operazione tra numeri interi senza segno, e' stato oltrepassato il confine **00000**, **99999** o in un verso o nell'altro; in caso contrario si ottiene **CF=0**.

3.2.3 Moltiplicazione tra numeri interi senza segno

Contrariamente a quanto si possa pensare, la moltiplicazione non crea alla **CPU** nessun problema legato ad un eventuale risultato (**prodotto**) di dimensioni superiori al massimo consentito; osserviamo infatti che:

***** per i numeri ad una cifra il prodotto massimo e':
`9 x 9 = 81` che e' minore di `10 x 10 = 100`
***** per i numeri a due cifre il prodotto massimo e':
`99 x 99 = 9801` che e' minore di `100 x 100 = 10000`
***** per i numeri a tre cifre il prodotto massimo e':
`999 x 999 = 998001` che e' minore di `1000 x 1000 = 1000000`
e cosi' via.

Possiamo dire quindi che moltiplicando tra loro due **fattori** formati ciascuno da una sola cifra, il prodotto massimo non puo' avere piu' di **2** cifre (**1+1**); moltiplicando tra loro due fattori formati ciascuno da due cifre, il prodotto massimo non puo' avere piu' di **4** cifre (**2+2**). Moltiplicando tra loro due fattori formati ciascuno da tre cifre, il prodotto massimo non puo' avere piu' di **6** cifre (**3+3**); moltiplicando tra loro due fattori formati ciascuno da quattro cifre, il prodotto massimo non puo' avere piu' di **8** cifre (**4+4**).
In generale, il prodotto tra due fattori formati ciascuno da **n** cifre, e' un numero che richiede al massimo **n+n=2n** cifre, cioe' il doppio di **n**; grazie a questa proprieta' delle moltiplicazioni tra numeri interi, la **CPU** puo' eseguire prodotti su numeri interi senza segno a **5** cifre, disponendo il risultato finale in due gruppi da **5** cifre ciascuno che uniti assieme forniscono il risultato completo a **10** cifre. La Figura 6 mostra alcuni esempi pratici:

Figura 6

```
1) 75922d x 44321d = 33649d 38962d
2) 02718d x 00024d = 00000d 65232d
3) 99999d x 99999d = 99998d 00001d
```

Le moltiplicazioni tra numeri interi a **5** cifre vengono eseguite direttamente (via hardware) dalla nostra **CPU** che e' dotata come abbiamo visto di architettura a **5** cifre; se vogliamo moltiplicare tra loro numeri formati da piu' di **5** cifre, dobbiamo procedere via software simulando sul computer lo stesso metodo che si segue usando carta e penna. Il meccanismo dovrebbe essere ormai chiaro; comunque, in un apposito capitolo verra' mostrato un programma di esempio. L'aspetto importante da ribadire ancora una volta, e' che la gestione di queste operazioni via software da parte del programmatore e' nettamente piu' lenta della gestione via hardware da parte della **CPU**.

3.2.4 Caso particolare per la moltiplicazione tra numeri interi senza segno

In relazione alle moltiplicazioni tra numeri interi senza segno, si presenta un caso particolare che ci permette di velocizzare notevolmente questa operazione sfruttando una proprieta' dei sistemi di numerazione posizionali; questo caso particolare e' rappresentato dalla eventualita' che uno dei due fattori sia esprimibile sotto forma di potenza con esponente intero della base **10**. Vediamo infatti quello che succede quando si moltiplica un numero intero senza segno per $\mathbf{10^n}$; per i numeri interi senza segno a **5** cifre, **n** puo' assumere uno tra i valori **1**, **2**, **3**, **4**.

350×10^1 = 350 x 10 = 00350d x 00010d = 00000d 03500d = 3500

Notiamo che la moltiplicazione ha determinato l'aggiunta di uno zero alla destra di **350**; tutto cio' equivale a far scorrere di un posto verso sinistra tutte le cifre di **350** riempiendo con uno zero il posto rimasto libero a destra.

350×10^2 = 350 x 100 = 00350d x 00100d = 00000d 35000d = 35000

Notiamo che la moltiplicazione ha determinato l'aggiunta di due zeri alla destra di **350**; tutto cio' equivale a far scorrere di due posti verso sinistra tutte le cifre di **350** riempiendo con due zeri i posti rimasti liberi a destra.

350×10^3 = 350 x 1000 = 00350d x 01000d = 00003d 50000d = 350000

Notiamo che la moltiplicazione ha determinato l'aggiunta di tre zeri alla destra di **350**; tutto cio' equivale a far scorrere di tre posti verso sinistra tutte le cifre di **350** riempiendo con tre zeri i posti rimasti liberi a destra.

350×10^4 = 350 x 10000 = 00350d x 10000d = 00035d 00000d = 3500000

Notiamo che la moltiplicazione ha determinato l'aggiunta di quattro zeri alla destra di **350**; tutto cio' equivale a far scorrere di quattro posti verso sinistra tutte le cifre di **350** riempiendo con quattro zeri i posti rimasti liberi a destra.

In definitiva, ogni volta che uno dei fattori e' esprimibile nella forma $\mathbf{10^n}$, e' possibile velocizzare notevolmente la moltiplicazione aggiungendo direttamente **n** zeri alla destra dell'altro fattore (facendo cosi' scorrere di **n** posti verso sinistra tutte le cifre dell'altro fattore); proprio per questo motivo, tutte le **CPU** forniscono una apposita istruzione per lo scorrimento verso sinistra delle cifre di un numero intero senza segno. Nel caso delle **CPU** della famiglia **80x86** questa istruzione viene chiamata **SHL** o **shift logical left** (scorrimento logico verso sinistra); e' sottinteso che questa istruzione debba essere usata con numeri interi senza segno.
Quando si utilizza questa istruzione, bisogna sempre ricordarsi del fatto che la **CPU** puo' gestire numeri interi senza segno aventi un numero limitato di cifre; utilizzando quindi un valore troppo alto per l'esponente **n**, si puo' provocare il "trabocco" da sinistra di cifre significative del risultato, ottenendo in questo modo un valore privo di senso. Se richiediamo ad esempio alla **CPU** lo scorrimento di **5** posti verso sinistra di tutte le cifre del numero **00350d**, otteniamo come risultato **00000d**!

3.2.5 Divisione tra numeri interi senza segno

Anche la divisione non crea problemi in quanto se si divide un numero (**dividendo**) per un altro numero (**divisore**), il risultato che si ottiene (**quoziente**) non potra' mai essere maggiore del dividendo; dividendo quindi tra loro due numeri interi senza segno a **5** cifre, il quoziente non potra' mai superare **99999**. Il caso peggiore che si puo' presentare e' infatti:
`99999 / 00001 = 99999`
Analogo discorso vale anche per il **resto** della divisione; come si sa' dalla matematica infatti, il resto di una divisione non puo' essere maggiore del divisore, e quindi non potra' mai superare **99999**.
La **CPU** anche in questo caso dispone il risultato in due gruppi da **5** cifre ciascuno; il primo gruppo contiene il quoziente mentre il secondo gruppo contiene il resto. Bisogna osservare infatti che in questo caso stiamo parlando di **divisione intera**, cioe' dell'operazione di divisione effettuata nell'insieme **N**; il quoziente e il resto che si ottengono devono appartenere a loro volta ad **N**, per cui devono essere numeri interi senza segno. Nel caso in cui il dividendo non sia un multiplo intero del divisore, si dovrebbe ottenere in teoria un numero con la virgola (numero reale); la **CPU** in questo caso tronchera' al quoziente la parte decimale, cioe' la parte posta dopo la virgola. La Figura 7 mostra una serie di esempi pratici; il simbolo **Q** indica il quoziente, mentre il simbolo **R** indica il resto.

Figura 7

```
1) 50000d / 25000d = [Q = 00002d, R = 00000d]
2) 32000d / 28000d = [Q = 00001d, R = 04000d]
3) 15000d / 20000d = [Q = 00000d, R = 15000d]
```

Come si puo' notare, trattandosi di divisione intera, se il dividendo e' piu' piccolo del divisore, si ottiene come risultato zero (esempio **3**).
Anche la divisione tra numeri con piu' di **5** cifre puo' essere simulata via software attraverso lo stesso procedimento che si segue con carta e penna; un esempio in proposito verra' mostrato in un apposito capitolo.

Quando si effettua una divisione, si presenta un caso particolare rappresentato dal divisore che vale zero; cosa succede se il divisore e' zero?
Come si sa' dalla matematica, dividendo un numero non nullo per zero, si ottiene un numero (in valore assoluto) infinitamente grande, tale cioe' da superare qualunque altro numero grande a piacere; naturalmente la **CPU** non e' in grado di gestire un numero infinitamente grande, per cui la divisione per zero viene considerata una condizione di errore (esattamente come accade con le calcolatrici). Si tratta di una situazione abbastanza delicata che generalmente viene gestita direttamente dal sistema operativo; il **DOS** ad esempio interrompe il programma in esecuzione, e stampa sullo schermo un messaggio del tipo:
`Divisione per zero`
Analogamente **Windows** mostra una finestra per informare che il programma in esecuzione verra' interrotto in seguito ad una:
`Operazione non valida`

3.2.6 Caso particolare per la divisione tra numeri interi senza segno

Anche per le divisioni tra numeri interi senza segno, si presenta il caso particolare citato prima per le moltiplicazioni; vediamo infatti quello che succede quando si divide un numero intero senza segno per $\mathbf{10^n}$ (con **n** compreso tra **1** e **4**).
85420 / 10^1 = 85420 / 10 = 85420d / 00010d = Q = 08542d, R = 00000d
Notiamo che la divisione ha determinato l'aggiunta di uno zero alla sinistra di **85420** facendo "traboccare" da destra la sua prima cifra; tutto cio' equivale a far scorrere di un posto verso destra tutte le cifre di **85420** riempiendo con uno zero il posto rimasto libero a sinistra. La cifra che

"trabocca" da destra rappresenta il resto (**0**) della divisione.
`85420 / 10`^{2}` = 85420 / 100 = 85420d / 00100d = Q = 00854d, R = 00020d`
Notiamo che la divisione ha determinato l'aggiunta di due zeri alla sinistra di **85420** facendo "traboccare" da destra le sue prime due cifre; tutto cio' equivale a far scorrere di due posti verso destra tutte le cifre di **85420** riempiendo con due zeri i posti rimasti liberi a sinistra. Le due cifre che "traboccano" da destra rappresentano il resto (**20**) della divisione.
`85420 / 10`^{3}` = 85420 / 1000 = 85420d / 01000d = Q = 00085d, R = 00420d`
Notiamo che la divisione ha determinato l'aggiunta di tre zeri alla sinistra di **85420** facendo "traboccare" da destra le sue prime tre cifre; tutto cio' equivale a far scorrere di tre posti verso destra tutte le cifre di **85420** riempiendo con tre zeri i posti rimasti liberi a sinistra. Le tre cifre che "traboccano" da destra rappresentano il resto (**420**) della divisione.
`85420 / 10`^{4}` = 85420 / 10000 = 85420d / 10000d = Q = 00008d, R = 05420d`
Notiamo che la divisione ha determinato l'aggiunta di quattro zeri alla sinistra di **85420** facendo "traboccare" da destra le sue prime quattro cifre; tutto cio' equivale a far scorrere di quattro posti verso destra tutte le cifre di **85420** riempiendo con quattro zeri i posti rimasti liberi a sinistra. Le quattro cifre che "traboccano" da destra rappresentano il resto (**5420**) della divisione.

In definitiva, ogni volta che il divisore e' esprimibile nella forma $\mathbf{10^n}$, e' possibile velocizzare notevolmente la divisione aggiungendo direttamente **n** zeri alla sinistra del dividendo (facendo cosi' scorrere di **n** posti verso destra tutte le cifre del dividendo); proprio per questo motivo, tutte le **CPU** forniscono una apposita istruzione per lo scorrimento verso destra delle cifre di un numero intero senza segno. Nel caso delle **CPU** della famiglia **80x86** questa istruzione viene chiamata **SHR** o **shift logical right** (scorrimento logico verso destra); e' sottinteso che questa istruzione debba essere usata con numeri interi senza segno.

3.3 Numeri interi con segno (positivi e negativi)

Qualunque informazione gestibile dal computer puo' essere codificata attraverso i numeri interi senza segno che abbiamo appena esaminato; questo sistema di codifica presenta pero' anche un interessantissimo "effetto collaterale" che permette alla **CPU** di gestire in modo diretto persino i numeri interi positivi e negativi (numeri con segno).
In matematica l'insieme numerico rappresentato dalla sequenza:
`..., -5, -4, -3, -2, -1, 0, +1, +2, +3, +4, +5, ...`
viene chiamato **insieme dei numeri interi relativi**, e viene indicato con il simbolo **Z**; come si puo' notare, l'insieme **N** e' incluso nell'insieme **Z**. Vediamo allora come bisogna procedere per codificare l'insieme **Z** sul nostro computer; anche in questo caso appare evidente il fatto che la **CPU** potra' gestire solo un sottoinsieme finito degli infiniti numeri appartenenti a **Z**.
Rispetto al caso dell'insieme **N**, la codifica dell'insieme **Z** non appare molto semplice perche' in questo caso dobbiamo rappresentare oltre ai numeri interi positivi e allo zero, anche i numeri negativi dotati cioe' di segno meno (**-**); il computer non ha la piu' pallida idea di cosa sia un segno piu' o un segno meno, ma ormai abbiamo capito che la soluzione consiste come al solito nel codificare qualunque informazione, compreso il segno di un numero, attraverso un numero stesso.
La prima considerazione da fare riguarda il fatto che come gia' sappiamo, l'architettura a **5** cifre della nostra **CPU** ci permette di rappresentare via hardware numeri interi compresi tra **00000d** e **99999d**, per un totale di **100000** possibili codici numerici; per motivi di simmetria nella codifica dell'insieme **Z**, questi **100000** codici differenti devono essere divisi in due parti uguali in modo da poter rappresentare **50000** numeri positivi (compreso lo zero) e **50000** numeri negativi. In sostanza, dovremmo essere in grado di rappresentare in questo modo un sottoinsieme finito di **Z** formato dai seguenti numeri relativi:
`-50000, -49999, ..., -3, -2, -1, +0, +1, +2, +3, ..., +49998, +49999`
Come si puo' notare, il numero **0** viene trattato come un numero positivo; proprio per questo motivo i numeri positivi arrivano solo a **+49999**.

Nella determinazione di questo metodo di codifica, fortunatamente ci viene incontro l'effetto collaterale a cui si e' accennato in precedenza; a tale proposito, torniamo per un momento alle sottrazioni tra numeri interi senza segno a **5** cifre, e osserviamo quello che accade in Figura 8:

Figura 8

```
00000d - 00001d = 99999d (CF = 1)
00000d - 00002d = 99998d (CF = 1)
00000d - 00003d = 99997d (CF = 1)
.................................
00000d - 49999d = 50001d (CF = 1)
00000d - 50000d = 50000d (CF = 1)
```

Come sappiamo, la **CPU** effettua queste sottrazioni segnalandoci un prestito attraverso **CF**; lasciamo perdere il prestito e osserviamo attentamente la Figura 8. Sembrerebbe che per la **CPU** **99999d** equivalga a **-1**, **99998d** equivalga a **-2**, **99997d** equivalga a **-3** e cosi' via; se questa deduzione fosse giusta, si otterrebbero quindi le corrispondenze mostrate in Figura 9:

Figura 9

```
49999d rappresenta il valore +49999
49998d rappresenta il valore +49998
...................................
00003d rappresenta il valore     +3
00002d rappresenta il valore     +2
00001d rappresenta il valore     +1
00000d rappresenta il valore     +0
99999d rappresenta il valore     -1
99998d rappresenta il valore     -2
99997d rappresenta il valore     -3
99996d rappresenta il valore     -4
...................................
50001d rappresenta il valore -49999
50000d rappresenta il valore -50000
```

Verifichiamo subito quello che accade provando a calcolare:
`1 - 1 = (+1) + (-1) = 0`
Con il metodo di codifica illustrato in Figura 9 si ottiene:
`00001d + 99999d = 00000d (CF=1)`
Lasciando perdere **CF** possiamo constatare che il risultato ottenuto (**00000d**) appare corretto.
Effettuiamo una seconda prova e calcoliamo:
`1 - 50000 = (+1) + (-50000) = -49999`
Con il metodo di codifica illustrato in Figura 9 si ottiene:
`00001d + 50000d = 50001d (CF=0)`
Lasciando perdere **CF** possiamo constatare che il risultato ottenuto e' giusto in quanto **50001d** e' proprio la codifica di **-49999**.
Effettuiamo una terza prova e calcoliamo:
`-1 - 2 = (-1) + (-2) = -3`
Con il metodo di codifica illustrato in Figura 9 si ottiene:
`99999d + 99998d = 99997d (CF=1)`
Lasciando perdere **CF** possiamo constatare che il risultato ottenuto e' giusto in quanto **99997d** e' proprio la codifica di **-3**.

Si tratta di un vero e proprio colpo di scena in quanto il metodo appena descritto sembra funzionare perfettamente!
Ed infatti, questo e' proprio il metodo che la **CPU** utilizza per gestire i numeri interi con segno; se proviamo ad effettuare qualsiasi altra prova, otteniamo sempre il risultato corretto. Anche il nostro

famoso contachilometri sembra confermare questi risultati; supponiamo infatti che il contachilometri segni **00000** e che possa funzionare anche al contrario quando la macchina va' in retromarcia. Se procediamo in marcia avanti il contachilometri segnera' via via:
`00001, 00002, 00003, 00004, 00005, .....`
come se volesse indicare:
`+1, +2, +3, +4, +5, ...`
Se procediamo ora in marcia indietro il contachilometri (partendo sempre da **00000**) segnera' via via:
`99999, 99998, 99997, 99996, 99995, .....`
come se volesse indicare:
`-1, -2, -3, -4, -5, ...`
Ovviamente scegliamo come confine tra numeri positivi e negativi quello delimitato da **49999** e **50000** perche' cosi' dividiamo i **100000** numeri possibili in due parti uguali; in questo modo, tutti i codici numerici a **5** cifre la cui cifra piu' significativa e' compresa tra **0** e **4**, rappresentano numeri interi positivi. Analogamente, tutti i codici numerici a **5** cifre la cui cifra piu' significativa e' compresa tra **5** e **9**, rappresentano numeri interi negativi; l'unico trascurabile difetto di questo sistema e' che lo zero appare come un numero positivo, mentre sappiamo che in matematica lo zero rappresenta il nulla e quindi non ha segno.
A questo punto ci chiediamo: come e' possibile che un sistema del genere possa funzionare? Naturalmente alla base di quello che abbiamo appena visto non ci sono fenomeni paranormali, ma bensi' la cosiddetta **aritmetica modulare**; per capire l'aritmetica modulare, facciamo ancora appello al nostro contachilometri supponendo che stia segnando **99998** (quindi abbiamo percorso **99998** Km). Se percorriamo altri **4** Km, il contachilometri segnera' via via:
`99998, 99999, 00000, 00001, 00002, .....`
invece di:
`99998, 99999, 100000, 100001, 100002, ......`
Il contachilometri infatti non puo' mostrare piu' di **5** cifre e quindi lavora in **modulo 100000** (modulo centomila) mostrando cioe' non i Km percorsi, ma il **resto** che si ottiene dalla divisione intera tra i Km percorsi e **100000**; infatti, indicando con **MOD** il resto della divisione intera e supponendo di partire da **00000** Km, si ottiene la situazione mostrata in Figura 10:

Figura 10

```
 00000 MOD 100000 = 00000
 00001 MOD 100000 = 00001
 00002 MOD 100000 = 00002
........................
 99998 MOD 100000 = 99998
 99999 MOD 100000 = 99999
100000 MOD 100000 = 00000
100001 MOD 100000 = 00001
100002 MOD 100000 = 00002
........................
199998 MOD 100000 = 99998
199999 MOD 100000 = 99999
200000 MOD 100000 = 00000
200001 MOD 100000 = 00001
200002 MOD 100000 = 00002
........................
```

In pratica, quando la nostra automobile supera i **99999** Km percorsi, il contachilometri si azzera; quando la nostra automobile supera i **199999** Km percorsi, il contachilometri si azzera una seconda volta. Quando la nostra automobile supera i **299999** Km percorsi, il contachilometri si azzera una terza volta; in generale, ogni volta che raggiungiamo un numero di Km percorsi che e' un multiplo intero di **100000**, il contachilometri si azzera e ricomincia a contare da **00000**.

Dalle considerazioni appena esposte si deduce quanto segue:

* se il contachilometri segna **00200**, e percorriamo **200** Km in marcia indietro, alla fine il contachilometri segnera' **00000**; possiamo dire quindi che:
`200 - 200 = 00000d`
* se il contachilometri segna **00200**, e percorriamo **201** Km in marcia indietro, alla fine il contachilometri segnera' non **-1** ma **99999**; possiamo dire quindi che:
`200 - 201 = 99999d`
* se il contachilometri segna **00200**, e percorriamo **4000** Km in marcia indietro, alla fine il contachilometri segnera' non **-3800** ma **96200**; possiamo dire quindi che:
`200 - 4000 = 96200d`
* se il contachilometri segna **00200**, e percorriamo **15000** Km in marcia indietro, alla fine il contachilometri segnera' non **-14800** ma **85200**; possiamo dire quindi che:
`200 - 15000 = 85200d`

Questa e' l'aritmetica modulare del nostro contachilometri a **5** cifre; tutto cio' si applica quindi in modo del tutto identico alla nostra **CPU** con architettura a **5** cifre.
Un'altra importante conseguenza legata alle cose appena viste, e' che nell'aritmetica modulo **100000**, possiamo dire che **00000** equivale a **100000**; possiamo scrivere quindi:

Figura 11

```
-1 = 0 - 1 = 100000 - 1 = 99999d
-2 = 0 - 2 = 100000 - 2 = 99998d
-3 = 0 - 3 = 100000 - 3 = 99997d
-4 = 0 - 4 = 100000 - 4 = 99996d
.............................................
```

3.3.1 Complemento a 10

Dalla Figura 11 ricaviamo subito la formula necessaria per convertire in modulo **100000** un numero intero negativo; in generale, dato un numero intero positivo compreso tra **+1** e **+49999**, per ottenere il suo equivalente negativo in modulo **100000** dobbiamo calcolare semplicemente:
`100000 - n`
Supponiamo ad esempio di voler ottenere la rappresentazione modulo **100000** di **-100**; applicando la formula precedente otteniamo:
`100000 - 100 = 99900d`
Nell'eseguire questa sottrazione con carta e penna, si presenta spesso la necessita' di richiedere dei prestiti; per evitare questo fastidio possiamo osservare quanto segue:
`100000 - n = (99999 + 1) - n = (99999 - n) + 1`
In questo caso il minuendo della sottrazione e' **99999**, per cui non sara' mai necessario richiedere dei prestiti; la tecnica appena descritta per la codifica dei numeri interi con segno viene chiamata **complemento a 10**.
Attraverso questa tecnica la **CPU** puo' **negare** facilmente un numero intero, cioe' cambiare di segno un numero intero; consideriamo ad esempio il numero **-25000** che in modulo **100000** viene rappresentato come:
`(99999 - 25000) + 1 = 74999 + 1 = 75000d`
E' chiaro allora che negando **75000** (cioe' **-25000**) dovremmo ottenere **+25000**; infatti:
`(99999 - 75000) + 1 = 24999 + 1 = 25000d`
che e' proprio la rappresentazione in complemento a **10** di **+25000**; questo significa che per la **CPU** si ha correttamente:
`-(-25000) = +25000`
In definitiva, con la tecnica appena vista la nostra **CPU** puo' gestire via hardware tutti i numeri relativi compresi tra **-50000** e **+49999**; la codifica utilizzata prende anche il nome di rappresentazione in complemento a **10** dei numeri con segno in modulo **100000**.

3.3.2 Addizione e sottrazione tra numeri interi con segno

Una volta definito il sistema di codifica in modulo **100000** per i numeri interi con segno, possiamo passare all'applicazione delle quattro operazioni fondamentali; in relazione all'addizione e alla sottrazione, si verifica subito una sorpresa estremamente importante. Riprendiamo a tale proposito l'esempio **3** di Figura 4:

```
50000d - 60000d = 90000d (CF = 1)
```

Osserviamo subito che se stiamo operando su numeri interi senza segno, la precedente operazione deve essere interpretata come:

```
50000 - 60000 = 90000
```

con prestito di **1** (**CF=1**); la presenza di un prestito ci indica come sappiamo che il risultato deriva da:

```
150000 - 60000 = 90000
```

Se invece stiamo operando su numeri interi con segno, osserviamo subito che **50000d** e' la codifica numerica di **-50000**, mentre **60000d** e' la codifica numerica di **-40000**; la precedente operazione deve essere quindi interpretata come:

```
(-50000) - (-40000) = -50000 + 40000 = -10000
```

Ma **-10000** in complemento a **10** si scrive proprio **90000d**; infatti:

```
100000 - 10000 = 90000d
```

L'aritmetica modulare ancora una volta e' responsabile di questa meraviglia che permette alla **CPU** di eseguire somme e sottrazioni senza la necessita' di distinguere tra numeri interi con o senza segno; per la **CPU** tutto questo significa una ulteriore semplificazione circuitale in quanto, in relazione alle addizioni e alle sottrazioni, si evitano tutte le complicazioni necessarie per distinguere tra numeri interi senza segno e numeri interi con segno. Nelle **CPU** della famiglia **80x86** troveremo quindi un'unica istruzione (chiamata **ADD**), per sommare numeri interi con o senza segno; analogamente, troveremo un'unica istruzione (chiamata **SUB**) per sottrarre numeri interi con o senza segno.

Naturalmente il programmatore puo' avere pero' la necessita' di distinguere tra numeri interi con o senza segno; in questo caso la **CPU** ci viene incontro mettendoci a disposizione oltre a **CF**, una serie di ulteriori flags attraverso i quali possiamo avere tutte le necessarie informazioni relative al risultato di una operazione appena eseguita. In sostanza, la **CPU** esegue addizioni e sottrazioni senza effettuare nessuna distinzione tra numeri interi con o senza segno; alla fine la **CPU** modifica una serie di flags che si riferiscono alcuni ai numeri senza segno, e altri ai numeri con segno. Sta al programmatore decidere quali flags consultare in base all'insieme numerico sul quale sta operando. Abbiamo gia' conosciuto in precedenza il flag **CF** che viene utilizzato per segnalare riporti nel caso delle addizioni, e prestiti nel caso delle sottrazioni; in relazione ai numeri con segno, facciamo la conoscenza con due nuovi flags chiamati **SF** e **OF**. Consideriamo prima di tutto il flag **SF** che rappresenta il cosiddetto **Sign Flag** (flag di segno); dopo ogni operazione la **CPU** pone **SF=0** se il risultato e' positivo o nullo, e **SF=1** se il risultato e' negativo. La Figura 12 mostra alcuni esempi pratici che chiariscono questi aspetti:

Figura 12

```
1) 30000d + 15000d = 45000d (CF = 0, SF = 0, OF = 0)
2) 95000d + 65000d = 60000d (CF = 1, SF = 1, OF = 0)
3) 40000d + 40000d = 80000d (CF = 0, SF = 1, OF = 1)
4) 60000d + 60000d = 20000d (CF = 1, SF = 0, OF = 1)
```

Nell'esempio **1**, **30000d** e **15000d** sono positivi sia nell'insieme dei numeri senza segno, sia nell'insieme dei numeri con segno; il risultato e' **45000d** che e' un numero positivo in entrambi gli insiemi. Per indicare che **45000d** e' positivo nell'insieme dei numeri con segno, la **CPU** pone **SF=0**; la **CPU** pone inoltre **CF=0** per indicare che nell'insieme dei numeri senza segno l'operazione non

provoca nessun riporto.
Nell'esempio **2**, **95000d** e **65000d** sono entrambi positivi nell'insieme dei numeri senza segno, ed entrambi negativi nell'insieme dei numeri con segno; il risultato e' **60000d** che e' positivo nel primo insieme e negativo nel secondo. Per indicare che **65000d** e' negativo nell'insieme dei numeri con segno, la **CPU** pone **SF=1**; la **CPU** pone inoltre **CF=1** per indicare che nell'insieme dei numeri senza segno l'operazione provoca un riporto.
Nell'esempio **3**, **40000d** e' positivo in entrambi gli insiemi numerici; il risultato e' **80000d** che e' positivo nel primo insieme e negativo nel secondo. Per indicare che **80000d** e' negativo nell'insieme dei numeri con segno, la **CPU** pone **SF=1**; la **CPU** pone inoltre **CF=0** per indicare che nell'insieme dei numeri senza segno l'operazione non provoca nessun riporto.
Nell'esempio **4**, **60000d** e' positivo nel primo insieme e negativo nel secondo; il risultato e' **20000d** che e' positivo in entrambi gli insiemi. Per indicare che **20000d** e' positivo nell'insieme dei numeri con segno, la **CPU** pone **SF=0**; la **CPU** pone inoltre **CF=1** per indicare che nell'insieme dei numeri senza segno l'operazione provoca un riporto.

Nel caso in cui si stia operando su numeri interi senza segno formati da sole **5** cifre ciascuno, possiamo dire che in Figura 12 gli esempi **1** e **3** producono un risultato valido cosi' com'e' (compreso cioe' tra **00000** e **99999**); gli esempi **2** e **4** producono invece un riporto (**CF=1**), per cui bisogna aggiungere un **1** alla sinistra delle **5** cifre del risultato.
Se stiamo operando invece su numeri interi senza segno aventi lunghezza arbitraria (formati cioe' da due o piu' gruppi di **5** cifre ciascuno), abbiamo gia' visto che dobbiamo sfruttare il contenuto di **CF** per proseguire l'operazione con le colonne successive alla prima; terminati i calcoli dobbiamo verificare il contenuto finale di **CF** per sapere se il risultato ottenuto e' valido cosi com'e' (**CF=0**) o se deve tenere conto del riporto (**CF=1**).

Passiamo ora al caso piu' impegnativo dei numeri interi con segno, e supponiamo inizialmente di operare su numeri formati da sole **5** cifre; al termine di ogni operazione possiamo consultare **SF** per conoscere il segno del risultato finale. Il metodo che utilizza la **CPU** per stabilire il segno di un numero e' molto semplice; abbiamo gia' visto infatti che nella rappresentazione in complemento a **10** dei numeri con segno in modulo **100000**, un numero e' positivo (**SF=0**) quando la sua cifra piu' significativa e' compresa tra **0** e **4**, ed e' invece negativo (**SF=1**) quando la sua cifra piu' significativa e' compresa tra **5** e **9**.
In relazione alla Figura 12 possiamo dire quindi che gli esempi **1** e **4** producono un risultato positivo (**SF=0**), mentre gli esempi **2** e **3** producono un risultato negativo (**SF=1**); osservando meglio la situazione ci accorgiamo pero' che c'e' qualcosa che non quadra. Nell'esempio **1** tutto fila liscio in quanto sommando tra loro due numeri positivi otteniamo come risultato un numero positivo; anche nell'esempio **2** tutto e' regolare in quanto sommando tra loro due numeri negativi otteniamo come risultato un numero negativo. Il discorso cambia invece nell'esempio **3** dove sommando tra loro due numeri positivi otteniamo come risultato un numero negativo; anche nell'esempio **4** vediamo che i conti non tornano in quanto sommando tra loro due numeri negativi otteniamo come risultato un numero positivo. Come si spiega questa situazione?
La risposta a questa domanda e' molto semplice; abbiamo visto in precedenza che nella rappresentazione in complemento a **10** dei numeri con segno in modulo **100000**, i numeri positivi vanno da **00000d** a **49999d**, mentre i numeri negativi vanno da **50000d** a **99999d**. E' chiaro allora che il risultato di una somma o di una sottrazione tra numeri con segno a **5** cifre, deve necessariamente rientrare in questi limiti; a questo punto possiamo capire quello che e' successo in Figura 12. Nell'esempio **1** abbiamo sommato i due numeri positivi **30000d** e **15000d** ottenendo correttamente un risultato positivo (**45000d**) compreso tra **00000d** e **49999d**; nell'esempio **2** abbiamo sommato i due numeri negativi **95000d** e **65000d** ottenendo correttamente un risultato negativo (**60000d**) compreso tra **50000d** e **99999d**. Nell'esempio **3** abbiamo sommato i due numeri positivi **40000d** e **40000d** ottenendo pero' un risultato negativo (**80000d**); questo risultato supera il

massimo consentito (**49999d**) per i numeri positivi a **5** cifre. Nell'esempio **4** abbiamo sommato i due numeri negativi **60000d** e **60000d** ottenendo pero' un risultato positivo (**20000d**); questo risultato e' inferiore al minimo consentito (**50000d**) per i numeri negativi a **5** cifre.

Riassumendo, nel caso dei numeri con segno a **5** cifre possiamo dire che sommando tra loro due numeri positivi, il risultato finale deve essere ancora un numero positivo non superiore a **+49999** (**49999d**); analogamente, sommando tra loro due numeri negativi, il risultato deve essere ancora un numero negativo non minore di **-50000** (**50000d**).
Se questo non accade vengono superati i limiti (inferiore o superiore), e si dice allora che si e' verificato un **trabocco**; come molti avranno intuito, la **CPU** ci informa dell'avvenuto trabocco attraverso un'altro flag che prende il nome di **Overflow Flag** (**OF**), cioe' **segnalatore di trabocco**.
Come funziona questo flag?
Lo si capisce subito osservando di nuovo la Figura 12 dove si nota che gli esempi **1** e **2** producono un risultato valido; in questo caso la **CPU** pone **OF=0**. La situazione cambia invece nell'esempio **3** dove la somma tra due numeri positivi produce un numero positivo troppo grande (si potrebbe dire "troppo positivo") che supera quindi il limite superiore di **49999d** e sconfina nell'area riservata ai numeri negativi; in questo caso la **CPU** ci segnala il trabocco ponendo **OF=1**. Anche nell'esempio **4** vediamo che la somma tra due numeri negativi produce un numero negativo troppo piccolo (si potrebbe dire "troppo negativo") che supera quindi il limite inferiore di **50000d** e sconfina nell'area riservata ai numeri positivi; anche in questo caso la **CPU** ci segnala il trabocco ponendo **OF=1**.
Sia nell'esempio **3** che nell'esempio **4** il segno del risultato e' l'opposto di quello che sarebbe dovuto essere; la **CPU** verifica il segno dei due numeri (**operandi**) prima di eseguire l'operazione, e se vede che il segno del risultato non e' corretto, pone **OF=1**.
Osserviamo che nel caso della somma tra un numero positivo e uno negativo, non si potra' mai verificare un trabocco; infatti, il caso peggiore che si puo' presentare e':

```
0 + (-50000) = 0 - 50000 = -50000
```

Nel nostro sistema di codifica l'operazione precedente diventa:

```
00000d + 50000d = 50000d
```

con **CF=0**, **SF=1** e **OF=0**.

Per capire ora il vero significato di **OF** analizziamo i valori assunti in sequenza da un numero a **5** cifre che viene via via incrementato di una unita' per volta (o che viene via via decrementato di una unita' per volta):

```
....., 49996, 49997, 49998, 49999, 50000, 50001, 50002, 50003, .....
```

In questa sequenza possiamo individuare un punto molto importante rappresentato dal "confine" che separa **49999** e **50000**; passando da **49999** a **50000** attraversiamo questo confine da sinistra a destra, mentre passando da **50000** a **49999** attraversiamo questo confine da destra a sinistra.
Se la somma tra due numeri positivi fornisce un risultato non superiore a **49999** (**SF=0**), vuol dire che siamo rimasti alla sinistra del confine (**OF=0**); se invece il risultato finale supera **49999** (**SF=1**), vuol dire che abbiamo oltrepassato il confine (**OF=1**) procedendo da sinistra a destra.
Se la somma tra due numeri negativi fornisce un risultato non inferiore a **50000** (**SF=1**), vuol dire che siamo rimasti alla destra del confine (**OF=0**); se invece il risultato finale e' inferiore a **50000** (**SF=0**), vuol dire che abbiamo oltrepassato il confine (**OF=1**) procedendo da destra a sinistra.
In definitiva possiamo dire che **OF=1** segnala il fatto che nel corso di una operazione tra numeri interi con segno, e' stato oltrepassato il confine **49999**, **50000** o in un verso o nell'altro; in caso contrario si ottiene **OF=0**.

Qualcuno abituato a lavorare con i linguaggi di alto livello, leggendo queste cose avra' la tentazione di scoppiare a ridere pensando che l'**Assembly** sia un linguaggio destinato a pazzi fanatici che perdono tempo con queste assurdita'; chi fa' questi ragionamenti pero' non si rende conto che questi problemi riguardano qualsiasi linguaggio di programmazione. Le considerazioni esposte in questo capitolo si riferiscono infatti al modo con cui la **CPU** gestisce le informazioni al suo interno; questo

modo di lavorare della **CPU** si ripercuote ovviamente su tutti i linguaggi di programmazione, compreso l'**Assembly**.
Si puo' citare un esempio famoso riguardante **Tetris** che assieme a **Packman** fu uno dei primi gloriosi giochi per computer comparsi sul mercato; **Tetris** era stato scritto in **BASIC**, e per memorizzare il punteggio veniva usato un tipo di dato **INTEGER**, cioe' un intero con segno che nelle architetture a **16** bit equivale al tipo **signed int** del **C** o al tipo **Integer** del **Pascal**. Come vedremo nel prossimo capitolo, una variabile di questo tipo puo' assumere tutti i valori compresi tra **-32768** e **+32767**; i giocatori piu' abili, riuscivano a superare abbondantemente i **30000** punti, e arrivati a **32767** bastava fare un'altro punto per veder comparire sullo schermo il punteggio di **-32768**!
Chi ha letto attentamente questo capitolo avra' gia' capito il perche'; chi invece prima stava ridendo adesso avra' modo di riflettere!

Torniamo ora alla nostra **CPU** per analizzare quello che succede nel momento in cui si vogliono eseguire via software, addizioni o sottrazioni su numeri interi con segno aventi ampiezza arbitraria (formati quindi da piu' di **5** cifre); in questo caso il metodo da applicare e' teoricamente molto semplice in quanto dobbiamo procedere esattamente come e' stato gia' mostrato in Figura 3 e in Figura 5 per i numeri interi senza segno. Una volta che l'operazione e' terminata, dobbiamo consultare non **CF** ma **OF** per sapere se il risultato e' valido (**OF=0**) o se si e' verificato un overflow (**OF=1**); se il risultato e' valido, possiamo consultare **SF** per sapere se il segno e' positivo (**SF=0**) o negativo (**SF=1**).
Per giustificare questo procedimento, e' necessario applicare tutti i concetti esposti in precedenza in relazione all'aritmetica modulare; osserviamo infatti che nel caso dei numeri interi con segno a **5** cifre, la **CPU** utilizza via hardware, la rappresentazione in complemento a **10** dei numeri in modulo **100000**. Che tipo di rappresentazione si deve utilizzare nel caso dei numeri interi con segno formati da piu' di **5** cifre?
E' chiaro che anche in questo caso, per coerenza con il sistema di codifica utilizzato dalla **CPU**, dobbiamo servirci ugualmente del complemento a **10**; e' necessario inoltre stabilire in anticipo il modulo dei numeri interi con segno sui quali vogliamo operare. Come abbiamo gia' visto in relazione ai numeri interi senza segno, e' opportuno scegliere sempre un numero di cifre che sia un multiplo intero di quello gestibile via hardware dalla **CPU** (nel nostro caso un multiplo intero di **5**); in questo modo e' possibile ottenere la massima efficienza possibile nell'algoritmo che esegue via software una determinata operazione matematica.
Supponiamo ad esempio di voler eseguire via software, addizioni e sottrazioni su numeri interi con segno a **10** cifre; in questo caso e' come se avessimo a che fare con un contachilometri a **10** cifre, e quindi il modulo di questi numeri e':
10^{10} = 10000000000
Ci troviamo quindi ad operare con una rappresentazione in complemento a **10** dei numeri interi con segno in modulo **10000000000**; dividiamo come al solito questi **10^{10}** numeri in due parti uguali in modo da poter codificare **5000000000** numeri interi positivi (compreso lo zero) e **5000000000** numeri interi negativi. In analogia a quanto abbiamo visto nel caso dei numeri interi con segno a **5** cifre, anche in questo caso, dato un numero intero positivo **n** compreso tra **+1** e **+4999999999**, il suo equivalente negativo si otterra' dalla formula:
`10000000000 - n = (9999999999 - n) + 1`
In base a questa formula si ottengono le corrispondenze mostrate in Figura 13

Figura 13
```
4999999999d rappresenta il valore +4999999999
4999999998d rappresenta il valore +4999999998
.............................................
0000000003d rappresenta il valore          +3
0000000002d rappresenta il valore          +2
```

```
0000000001d rappresenta il valore         +1
0000000000d rappresenta il valore         +0
9999999999d rappresenta il valore         -1
9999999998d rappresenta il valore         -2
9999999997d rappresenta il valore         -3
9999999996d rappresenta il valore         -4
.........................................
5000000001d rappresenta il valore -4999999999
5000000000d rappresenta il valore -5000000000
```

Come possiamo notare dalla Figura 13, si tratta di una situazione sostanzialmente identica a quella gia' vista per i numeri interi con segno a **5** cifre; anche in questo caso quindi, il segno del numero e' codificato nella sua cifra piu' significativa. Tutte le codifiche numeriche la cui cifra piu' significativa e' compresa tra **0** e **4** rappresentano numeri interi positivi (compreso lo zero); tutte le codifiche numeriche la cui cifra piu' significativa e' compresa tra **5** e **9** rappresentano numeri interi negativi.
In definitiva, se il programmatore vuole eseguire addizioni o sottrazioni su numeri interi con segno formati al massimo da **10** cifre, deve prima di tutto convertire questi numeri nella rappresentazione in complemento a **10** modulo **10000000000**; a questo punto si puo' eseguire l'addizione o la sottrazione con lo stesso metodo gia' visto in Figura 3 e in Figura 5. La Figura 14 illustra alcuni esempi pratici (i numeri interi con segno sono gia' stati convertiti nella rappresentazione di Figura 13):

Figura 14

```
1) 20999d 48891d + 11240d 66666d = ?

   colonna 1: 48891d + 66666d = 15557d      (CF = 1)
   colonna 2: 20999d + 11240d + CF = 32240d (CF = 0, SF = 0, OF = 0)
   risultato finale: 32240d 15557d

2) 90400d 22760d + 87500d 32888d = ?

   colonna 1: 22760d + 32888d = 55648d      (CF = 0)
   colonna 2: 90400d + 87500d + CF = 77900d (CF = 1, SF = 1, OF = 0)
   risultato finale: 77900d 55648d

3) 55260d 11345d + 60200d 23150d = ?

   colonna 1: 11345d + 23150d = 34495d      (CF = 0)
   colonna 2: 55260d + 60200d + CF = 15460d (CF = 1, SF = 0, OF = 1)
   risultato finale: 15460d 34495d

4) 40200d 77921d + 39800d 55480d = ?

   colonna 1: 77921d + 55480d = 33401d      (CF = 1)
   colonna 2: 40200d + 39800d + CF = 80001d (CF = 0, SF = 1, OF = 1)
   risultato finale: 80001d 33401d
```

Se stiamo operando sui numeri interi senza segno, tutte le codifiche numeriche mostrate in Figura 14 si riferiscono ovviamente a numeri positivi a **10** cifre (compreso lo zero); in questo caso, al termine dell'operazione consultiamo **CF** per sapere se c'e' stato o meno un riporto finale.
Se stiamo operando sui numeri interi con segno, tutte le codifiche numeriche mostrate in Figura 14 si riferiscono ovviamente a numeri a **10** cifre sia positivi (compreso lo zero) che negativi, rappresentati in complemento a **10**; l'operazione si svolge esattamente come per i numeri interi senza segno, verificando attraverso **CF** la presenza di eventuali riporti (per l'addizione) o prestiti (per la sottrazione). Terminata l'operazione dobbiamo consultare non **CF** ma **OF** per sapere se il

risultato e' valido o se c'e' stato un overflow; e' chiaro infatti che solo l'ultima colonna contiene la cifra che codifica il segno.
Supponendo quindi di operare sui numeri interi con segno, l'esempio **1** deve essere interpretato in questo modo:
`(+2099948891) + (+1124066666) = +3224015557`
L'operazione viene svolta in modo corretto in quanto la somma di questi due numeri positivi e' un numero positivo non superiore a **+4999999999**; la **CPU** ci dice infatti che il risultato e' valido (**OF=0**) ed e' positivo (**SF=0**).
L'esempio **2** deve essere interpretato in questo modo:
`(-959977240) + (-1249967112) = -2209944352`
L'operazione viene svolta in modo corretto in quanto la somma di questi due numeri negativi e' un numero negativo non inferiore a **-5000000000**; osserviamo infatti che il numero negativo **-2209944352** in complemento a **10** modulo **10000000000** si scrive:
`10000000000 - 2209944352 = 7790055648`
La **CPU** ci dice che il risultato e' valido (**OF=0**) ed e' negativo (**SF=1**).
L'esempio **3** deve essere interpretato in questo modo:
`(-4473988655) + (-3979976850) = +1546034495`
L'operazione viene svolta in modo sbagliato in quanto la somma di questi due numeri negativi e' un numero negativo inferiore a **-5000000000** che sconfina nell'insieme dei numeri positivi; la **CPU** ci dice infatti che si e' verificato un overflow (**OF=1**) in quanto sommando due numeri negativi e' stato ottenuto un numero positivo (**SF=0**).
L'esempio **4** deve essere interpretato in questo modo:
`(+4020077921) + (+3980055480) = -1999866599`
L'operazione viene svolta in modo sbagliato in quanto la somma di questi due numeri positivi e' un numero positivo superiore a **+4999999999** che sconfina nell'insieme dei numeri negativi; la **CPU** ci dice infatti che si e' verificato un overflow (**OF=1**) in quanto sommando due numeri positivi e' stato ottenuto un numero negativo (**SF=1**).

3.3.3 Moltiplicazione tra numeri interi con segno

Come sappiamo dalla matematica, moltiplicando tra loro due numeri interi con segno si ottiene un risultato il cui segno viene determinato con le seguenti regole:

(+) * (+) = (+)
(+) * (-) = (-)
(-) * (+) = (-)
(-) * (-) = (+)

Osserviamo inoltre che in valore assoluto il risultato che si ottiene e' assolutamente indipendente dal segno; abbiamo ad esempio:
`(+350) x (+200) = +70000` e `(-350) x (+200) = -70000`
In valore assoluto si ottiene **+70000** in entrambi i casi.
In base a queste considerazioni, possiamo dedurre il metodo seguito dalla nostra **CPU** per eseguire via hardware moltiplicazioni tra numeri con segno a **5** cifre (espressi naturalmente in complemento a **10** modulo **100000**); le fasi che vengono svolte sono le seguenti:

***** la **CPU** converte se necessario i due fattori in numeri positivi;
***** la **CPU** esegue la moltiplicazione tra numeri positivi;
***** la **CPU** aggiunge il segno al risultato in base alle regole esposte in precedenza.

Prima di analizzare alcuni esempi pratici, e' necessario ricordare che moltiplicando tra loro due numeri interi a **n** cifre, si ottiene un risultato avente al massimo **2n** cifre; possiamo dire quindi che

in relazione ai numeri interi con segno, la moltiplicazione produce un effetto molto importante che consiste in un cambiamento di modulo. Nel caso ad esempio della nostra **CPU**, possiamo notare che moltiplicando tra loro due numeri interi a **5** cifre, otteniamo un risultato a **10** cifre; per i numeri interi con segno questo significa che stiamo passando dal modulo **100000** al modulo **10000000000**. E' chiaro quindi che al momento di aggiungere il segno al risultato della moltiplicazione, la nostra **CPU** deve convertire il risultato stesso in un numero intero con segno espresso in modulo **10000000000**.
Questa situazione non puo' mai produrre un overflow; osserviamo infatti che in modulo **10000000000** i numeri negativi vanno da **-5000000000** a **-1**, e i numeri positivi vanno da **+0** a **+4999999999**. Moltiplicando tra loro due numeri interi con segno a **5** cifre, il massimo risultato positivo ottenibile e':
`(-50000) x (-50000) = +2500000000`
che e' nettamente inferiore a **+4999999999**.
Moltiplicando tra loro due numeri interi con segno a **5** cifre, il minimo risultato negativo ottenibile e':
`(+49999) x (-50000) = -2499950000`
che e' nettamente superiore a **-5000000000**.

Vediamo ora alcuni esempi pratici che hanno anche lo scopo di evidenziare la differenza che esiste tra la moltiplicazione di numeri interi senza segno e la moltiplicazione di numeri interi con segno; questa differenza dipende proprio dal cambiamento di modulo provocato da questo operatore matematico.

1) Vogliamo calcolare:
`(+33580) x (+41485) = +1393066300`
Nell'insieme dei numeri senza segno, si ha la seguente codifica:
`33580d x 41485d = 1393066300d`
Nell'insieme dei numeri con segno, si ha la seguente codifica:
`33580d x 41485d = 1393066300d`
I risultati che si ottengono nei due casi sono identici in quanto le due codifiche **33580d** e **41485d** rappresentano numeri positivi sia nell'insieme degli interi senza segno sia nell'insieme degli interi con segno.

2) Vogliamo calcolare:
`(-11100) x (+10000) = -111000000`
La codifica numerica di **-11100** e' **88900d**, mentre la codifica numerica di **+10000** e' **10000d**; nell'insieme dei numeri interi senza segno si ottiene:
`88900d x 10000d = 0889000000d`
Nell'insieme dei numeri interi con segno la **CPU** nota che **88900d** e' un numero negativo (**-11100**) per cui lo converte nel suo equivalente positivo attraverso la seguente negazione:
`100000d - 88900d = 11100d`
che e' proprio la rappresentazione in complemento a **10** di **+11100**; a questo punto la **CPU** svolge la moltiplicazione ottenendo:
`11100d x 10000d = 0111000000d`
In base al fatto che **meno per piu' = meno**, la **CPU** deve negare il risultato ottenendo la rappresentazione in complemento a **10** di **-111000000**; in modulo **10000000000** si ha quindi:
`10000000000d - 111000000d = 9889000000d`
Come si puo' notare, nell'insieme dei numeri interi senza segno si ottiene **0889000000d**, mentre nell'insieme dei numeri interi con segno si ottiene **9889000000d**; questi due risultati sono completamente diversi tra loro, e questo significa che la nostra **CPU** nell'eseguire una moltiplicazione ha bisogno di sapere se vogliamo operare sui numeri interi senza segno o sui numeri interi con segno. Questa e' una diretta conseguenza del fatto che la moltiplicazione provoca un cambiamento di modulo; l'addizione e la sottrazione non provocano nessun cambiamento di

modulo, per cui la **CPU** non ha in questo caso la necessita' di distinguere tra numeri interi senza segno e numeri interi con segno. Nel caso ad esempio delle **CPU** della famiglia **80x86**, l'istruzione per le moltiplicazioni tra numeri interi senza segno viene chiamata **MUL**; l'istruzione per le moltiplicazioni tra numeri interi con segno viene invece chiamata **IMUL**.

Se vogliamo moltiplicare tra loro numeri interi con segno formati da piu' di **5** cifre, dobbiamo procedere come al solito via software simulando lo stesso procedimento che si segue con carta e penna; naturalmente in questo caso dobbiamo anche applicare le considerazioni appena esposte in relazione al cambiamento di modulo. Per poter svolgere via software questa operazione il programmatore deve quindi scrivere anche le istruzioni necessarie per la negazione di numeri interi con segno formati da piu' di **5** cifre; in un apposito capitolo vedremo un esempio pratico.

3.3.4 Caso particolare per la moltiplicazione tra numeri interi con segno

Abbiamo gia' visto che per le moltiplicazioni tra numeri interi senza segno si presenta un caso particolare rappresentato dalla eventualita' che uno dei due fattori sia esprimibile sotto forma di potenza con esponente intero della base **10**; tutte le considerazioni esposte per i numeri interi senza segno si applicano anche ai numeri interi con segno. Le **CPU** della famiglia **80x86** forniscono una apposita istruzione chiamata **SAL** o **shift arithmetic left** (scorrimento aritmetico verso sinistra); e' sottinteso che questa istruzione debba essere utilizzata con numeri interi con segno. Osserviamo pero' che a causa del fatto che il segno di un numero viene ricavato dalla sua cifra piu' significativa (quella piu' a sinistra), gli effetti prodotti dall'istruzione **SAL** sono identici agli effetti prodotti dall'istruzione **SHL** per cui queste due istruzioni sono interscambiabili.
Consideriamo ad esempio il codice numerico **99325d** che per i numeri interi senza segno rappresenta **99325**, mentre per i numeri interi con segno rappresenta **-675**; utilizziamo ora l'istruzione **SHL** per lo scorrimento di un posto verso sinistra delle cifre di **99325d**. In questo modo otteniamo **93250d**; applicando l'istruzione **SAL** a **99325d** si ottiene lo stesso identico risultato.
Per l'istruzione **SAL** valgono le stesse avvertenze gia' discusse per **SHL**; anche con **SAL** quindi, con un eccessivo scorrimento di cifre verso sinistra, si corre il rischio di perdere cifre significative del risultato; nel caso poi dei numeri interi con segno, si rischia anche di perdere la cifra piu' significativa che codifica il segno del numero stesso.
Consideriamo ad esempio il codice **99453** che rappresenta **-547**; se usiamo **SAL** (o **SHL**) per far scorrere di un posto verso sinistra le cifre di **99453d** otteniamo **94530d**. In complemento a **10** questa e' la codifica di:

```
-(100000 - 94530) = -5470 = (-547 x 10)
```

Il risultato quindi e' giusto in quanto abbiamo moltiplicato **-547** per **10**.
Se usiamo ora **SAL** per far scorrere di due posti verso sinistra le cifre di **99453d** otteniamo **45300d**; in complemento a **10** questa e' la codifica di **+45300** che non ha niente a che vedere con il risultato che ci attendevamo.

3.3.5 Divisione tra numeri interi con segno

Come sappiamo dalla matematica, dividendo tra loro due numeri interi con segno si ottengono un quoziente ed un resto i cui segni vengono determinati con le seguenti regole:

(+) / (+) = quoziente positivo e resto positivo;
(+) / (-) = quoziente negativo e resto positivo;
(-) / (+) = quoziente negativo e resto negativo;
(-) / (-) = quoziente positivo e resto negativo.

Osserviamo inoltre che in valore assoluto il quoziente e il resto che si ottengono sono assolutamente

indipendenti dal segno; abbiamo ad esempio:
`(+350) / (+200) = Q = +1, R = +150` e `(-350) / (+200) = Q = -1, R = -150`
In valore assoluto si ottiene **Q = +1, R = +150** in entrambi i casi.
In base a queste considerazioni, possiamo dedurre il metodo seguito dalla nostra **CPU** per eseguire via hardware divisioni tra numeri interi con segno a **5** cifre (espressi naturalmente in complemento a **10** modulo **100000**); le fasi che vengono svolte sono le seguenti:

***** la **CPU** converte se necessario il dividendo e il divisore in numeri positivi;
***** la **CPU** esegue la divisione tra numeri positivi;
***** la **CPU** aggiunge il segno al quoziente e al resto in base alle regole esposte in precedenza.

Prima di analizzare alcuni esempi pratici, e' necessario ricordare che la divisione intera tra numeri interi a **n** cifre, produce un quoziente intero e un resto intero aventi al massimo **n** cifre; nel caso ad esempio della nostra **CPU**, dividendo tra loro due numeri interi a **5** cifre, otteniamo un quoziente a **5** cifre e un resto a **5** cifre. Se si sta operando sui numeri interi senza segno, bisogna anche tenere presente che sia il quoziente sia il resto, devono essere compresi tra **-50000** e **+49999**; se si ottiene un quoziente minore di **-50000** o maggiore di **+49999**, la **CPU** crede di trovarsi nella stessa situazione della divisione per zero (quoziente troppo grande), per cui il **SO** interrompe il nostro programma e visualizza un messaggio di errore. Con i numeri interi con segno a **5** cifre, questa situazione si verifica nel seguente caso particolare:
`(-50000) / (-1) = Q = +50000, R = 0`
Per i numeri interi con segno a **5** cifre, **+50000** supera il massimo valore positivo consentito **+49999**; in tutti gli altri casi, l'overflow e' impossibile.

Vediamo ora alcuni esempi pratici che hanno anche lo scopo di evidenziare la differenza che esiste tra la divisione di numeri interi senza segno e la divisione di numeri interi con segno; questa differenza dipende dal fatto che come vedremo in un prossimo capitolo, la **CPU** calcola la divisione attraverso un metodo che provoca un cambiamento di modulo.

1) Vogliamo calcolare:
`(+31540) / (+6728) = Q = +4, R = +4628`
Nell'insieme dei numeri senza segno, si ha la seguente codifica:
`31540d / 06728d = Q = 00004d, R = 04628d`
Nell'insieme dei numeri con segno, si ha la seguente codifica:
`31540d / 06728d = Q = 00004d, R = 04628d`
I risultati che si ottengono nei due casi sono identici in quanto le due codifiche **31540d** e **06728d** rappresentano numeri positivi sia nell'insieme degli interi senza segno sia nell'insieme degli interi con segno.

2) Vogliamo calcolare:
`(-30400) / (+10000) = Q = -3, R = -400`
La codifica numerica di **-30400** e' **69600d**, mentre la codifica numerica di **+10000** e' **10000d**; nell'insieme dei numeri interi senza segno si ottiene:
`69600d / 10000d = Q = 00006d, R = 09600d`
Nell'insieme dei numeri interi con segno la **CPU** nota che **69600d** e' un numero negativo (**-30400**) per cui lo converte nel suo equivalente positivo attraverso la seguente negazione:
`100000d - 69600d = 30400d`
che e' proprio la rappresentazione in complemento a **10** di **+30400**; a questo punto la **CPU** svolge la divisione ottenendo:
`30400d / 10000d = Q = 00003d, R = 00400d`
In base al fatto che **meno diviso piu' = quoziente negativo e resto negativo**, la **CPU** deve negare sia il quoziente sia il resto; per il quoziente si ha:
`100000d - 00003d = 99997d`

Per il resto si ha:
```
100000d - 00400d = 99600d
```
Come si puo' notare, nell'insieme dei numeri interi senza segno si ottiene **Q=00006d** e **R=09600d**, mentre nell'insieme dei numeri interi con segno si ottiene **Q=99997d** e **R=99600d**; questi due risultati sono completamente diversi tra loro, e questo significa che la nostra **CPU** nell'eseguire una divisione ha bisogno di sapere se vogliamo operare sui numeri interi senza segno o sui numeri interi con segno. Nel caso ad esempio delle **CPU** della famiglia **80x86**, l'istruzione per le divisioni tra numeri interi senza segno viene chiamata **DIV**; l'istruzione per le divisioni tra numeri interi con segno viene invece chiamata **IDIV**.

Se vogliamo dividere tra loro numeri interi con segno formati da piu' di **5** cifre, dobbiamo procedere come al solito via software simulando lo stesso procedimento che si segue con carta e penna; naturalmente in questo caso dobbiamo anche applicare le considerazioni appena esposte in relazione al cambiamento di modulo. Per poter svolgere via software questa operazione il programmatore deve quindi scrivere anche le istruzioni necessarie per la negazione di numeri interi con segno formati da piu' di **5** cifre; in un apposito capitolo vedremo un esempio pratico.

3.3.6 Caso particolare per la divisione tra numeri interi con segno

Abbiamo gia' visto che per le divisioni tra numeri interi senza segno si presenta un caso particolare rappresentato dalla eventualita' che il divisore sia esprimibile sotto forma di potenza con esponente intero della base **10**; tutte le considerazioni esposte per i numeri interi senza segno valgono anche per i numeri interi con segno. Le **CPU** della famiglia **80x86** forniscono una apposita istruzione chiamata **SAR** o **shift arithmetic right** (scorrimento aritmetico verso destra); e' sottinteso che questa istruzione debba essere utilizzata con numeri interi con segno.
Questa volta pero' le due istruzioni **SHR** e **SAR** non sono interscambiabili in quanto producono effetti molto differenti; l'istruzione **SAR** infatti ha il compito di far scorrere verso destra le cifre di un numero intero con segno, provvedendo anche a preservare il segno del risultato.
Consideriamo ad esempio il codice numerico **00856d** che rappresenta il valore **+856** sia per i numeri interi senza segno sia per i numeri interi con segno; utilizziamo l'istruzione **SHR** per lo scorrimento di un posto verso destra delle cifre di **00856d**. In questo modo otteniamo **00085d**; come si puo' notare, **SHR** ha diviso per **10** il numero positivo **856** ottenendo correttamente il quoziente **85**.
Utilizziamo ora l'istruzione **SAR** per lo scorrimento di un posto verso destra delle cifre di **00856d**; anche in questo caso otteniamo **00085d**, e cioe' il quoziente **85**. L'istruzione **SAR** infatti constatando che **00856d** e' un numero positivo, ha aggiunto uno zero alla sua sinistra in modo da ottenere un risultato ancora positivo.
Consideriamo ora il codice numerico **99320d** che per i numeri interi senza segno rappresenta **99320**, mentre per i numeri interi con segno rappresenta **-680**; utilizziamo l'istruzione **SHR** per lo scorrimento di un posto verso destra delle cifre di **99320d**. In questo modo otteniamo **09932d**; come si puo' notare, **SHR** ha diviso per **10** il numero positivo **99320** ottenendo correttamente il quoziente **9932**.
Utilizziamo ora l'istruzione **SAR** per lo scorrimento di un posto verso destra delle cifre di **99320d**; possiamo cosi' constatare che questa volta si ottiene **99932d** che e' la codifica di:
```
-(100000 - 99932) = -68 = (-680) / 10
```
Anche in questo caso il risultato e' corretto in quanto **SAR**, constatando che **99320d** e' un numero negativo, ha aggiunto un **9** e non uno **0** alla sua sinistra in modo da ottenere un risultato ancora negativo; se utilizziamo quindi **SHR** con i numeri interi con segno otteniamo risultati privi di senso.
I concetti appena esposti sull'estensione del segno di un numero, vengono chiariti nel seguito del capitolo.
Osserviamo infine che in certi casi **SAR** fornisce un risultato sbagliato; il perche' verra' spiegato in un altro capitolo nel quale si parlera' dettagliatamente delle istruzioni di scorrimento.

3.4 Estensione del segno

Sempre in relazione ai numeri interi con segno, e' necessario esaminare una situazione molto delicata che si verifica nel momento in cui abbiamo la necessita' di effettuare un cambiamento di modulo; in sostanza, dobbiamo analizzare quello che succede quando vogliamo passare da modulo **m** a modulo **n**.
Supponiamo ad esempio di voler passare da modulo **m=100000** a modulo **n=10000000000**; vogliamo cioe' convertire i numeri interi a **5** cifre in numeri interi a **10** cifre.
Nel caso dei numeri interi senza segno la soluzione e' semplicissima; osserviamo infatti che con **10** cifre possiamo codificare tutti i numeri interi senza segno compresi tra **0** e **+9999999999**; otteniamo quindi le corrispondenze mostrate in Figura 15:

Figura 15

```
9999999999d rappresenta il valore +9999999999
9999999998d rappresenta il valore +9999999998
.............................................
0000099999d rappresenta il valore      +99999
0000099998d rappresenta il valore      +99998
.............................................
0000000003d rappresenta il valore          +3
0000000002d rappresenta il valore          +2
0000000001d rappresenta il valore          +1
0000000000d rappresenta il valore          +0
```

Possiamo dire quindi che nel passaggio da modulo **m=100000** a modulo **n=10000000000**, il codice **00000d** diventa **0000000000d**, il codice **00005d** diventa **0000000005d**, il codice **03850d** diventa **0000003850d**, il codice **88981d** diventa **0000088981d** e cosi' via; in sostanza, i codici a **5** cifre mostrati in Figura 1 vengono convertiti in codici a **10** cifre aggiungendo **5** zeri alla sinistra di ciascuno di essi.

Passiamo ora al caso piu' delicato dei numeri interi con segno; come si puo' facilmente intuire, in questo caso non dobbiamo fare altro che estendere al modulo **n=10000000000** tutti i concetti gia' esposti per il modulo **m=100000**. Prima di tutto dividiamo i $\mathbf{10^{10}}$ possibili codici numerici in due parti uguali; a questo punto, con i codici compresi tra **0000000000d** e **4999999999d** possiamo rappresentare tutti i numeri interi positivi compresi tra **0** e **+4999999999**. Con i codici numerici compresi tra **5000000000d** e **9999999999d** possiamo rappresentare tutti i numeri interi negativi compresi tra **-5000000000** e **-1**; le corrispondenze che si ottengono vengono mostrate in Figura 16:

Figura 16

```
4999999999d rappresenta il valore +4999999999
4999999998d rappresenta il valore +4999999998
.............................................
0000049999d rappresenta il valore      +49999
0000049998d rappresenta il valore      +49998
.............................................
0000000003d rappresenta il valore          +3
0000000002d rappresenta il valore          +2
0000000001d rappresenta il valore          +1
0000000000d rappresenta il valore          +0
9999999999d rappresenta il valore          -1
9999999998d rappresenta il valore          -2
9999999997d rappresenta il valore          -3
9999999996d rappresenta il valore          -4
.............................................
9999950001d rappresenta il valore      -49999
```

```
9999950000d rappresenta il valore      -50000
...........................................
5000000001d rappresenta il valore -4999999999
5000000000d rappresenta il valore -5000000000
```

Possiamo constatare quindi che nel passaggio da modulo **m=100000** a modulo **n=10000000000**, i codici numerici a **5** cifre compresi tra **00000d** e **49999d** (che rappresentano numeri interi positivi), vengono trasformati nei corrispondenti codici numerici compresi tra **0000000000d** e **0000049999d**; questo significa che tutti i numeri interi positivi a **5** cifre vengono convertiti in numeri interi positivi a **10** cifre aggiungendo **00000** alla loro sinistra.
Possiamo constatare inoltre che nel passaggio da modulo **m=100000** a modulo **n=10000000000**, i codici numerici a **5** cifre compresi tra **50000d** e **99999d** (che rappresentano numeri interi negativi), vengono trasformati nei corrispondenti codici numerici compresi tra **9999950000d** e **9999999999d**; questo significa che tutti i numeri interi negativi a **5** cifre vengono convertiti in numeri interi negativi a **10** cifre aggiungendo **99999** alla loro sinistra.
Queste importantissime considerazioni rappresentano il concetto fondamentale di **estensione del segno** di un numero intero con segno; tutte le cose appena viste possono essere dimostrate anche con il complemento a **10** applicato ai numeri interi con segno in modulo **10000000000**.
Il numero negativo **-3** in modulo **10000000000** diventa:
`10000000000 - 3 = 9999999997d`
Il numero negativo **-450** in modulo **10000000000** diventa:
`10000000000 - 450 = 9999999550d`
Il numero negativo **-10998** in modulo **10000000000** diventa:
`10000000000 - 10998 = 9999989002d`
Il numero negativo **-50000** in modulo **10000000000** diventa:
`10000000000 - 50000 = 9999950000d`

L'aritmetica modulare permette quindi alla **CPU** di estendere facilmente il segno di un numero intero con segno; nel caso di un numero intero positivo l'estensione del segno consiste nell'aggiunta di zeri alla sinistra del numero stesso, mentre nel caso di un numero intero negativo l'estensione del segno consiste nell'aggiunta di una serie di **9** alla sinistra del numero stesso.

Un'ultima considerazione riguarda il fatto che nel cambiamento di modulo generalmente ha senso solo il passaggio da modulo **m** a modulo **n** con **n** maggiore di **m** (che e' il caso appena esaminato); infatti, se **n** e' minore di **m**, si puo' verificare una perdita di cifre significative nella rappresentazione dei numeri. Supponiamo ad esempio di lavorare in modulo **m=10000000000** e di voler passare al modulo **n=100000**; in modulo **m**, il numero negativo **-1458563459** si codifica come:
`10000000000 - 1458563459 = 8541436541d`
Per convertire questo codice numerico in modulo **n=100000**, dovremmo troncare le sue **5** cifre piu' significative ottenendo cosi' il codice **36541d**; ma in modulo **n=100000** questa e' la codifica del numero positivo **+36541**. Come si puo' notare, non solo abbiamo perso **5** cifre significative del numero originario, ma siamo anche passati da un numero negativo ad un numero positivo; questo e' proprio quello che succede con i linguaggi di alto livello come il **C** quando ad esempio si copia il contenuto di un **long** (intero con segno a **32** cifre) in un **int** (intero con segno a **16** cifre).

3.5 Equivalenza tra le quattro operazioni

Supponiamo di avere una **CPU** capace di eseguire solamente addizioni, negazioni e scorrimenti di cifre sui numeri interi; una **CPU** di questo genere e' perfettamente in grado di eseguire anche sottrazioni, moltiplicazioni e divisioni.
Cominciamo con l'osservare che dati due numeri interi **A** e **B** positivi o negativi, possiamo scrivere:
`A - B = (+A) + (-B)`
In sostanza, la sottrazione tra **A** e **B** equivale alla somma tra **A** e l'opposto di **B**; la nostra **CPU** quindi puo' trasformare la differenza tra due numeri nella somma tra il primo numero e la negazione del secondo numero.
In relazione alla moltiplicazione, dati due numeri interi **A** e **B** positivi o negativi, il prodotto tra **A** e **B** in valore assoluto e' pari ad **A** volte **B** oppure a **B** volte **A**; anche la moltiplicazione tra numeri interi puo' essere quindi convertita in una serie di addizioni. Nel caso ad esempio di **A=3** e **B=6** possiamo scrivere:
`A x B = 3 x 6 = 3 + 3 + 3 + 3 + 3 + 3 = 6 + 6 + 6 = 18`
Se uno o entrambi i fattori sono negativi, la **CPU** puo' seguire il metodo gia' illustrato in precedenza per la moltiplicazione tra numeri interi con segno; la nostra **CPU** puo' quindi effettuare anche moltiplicazioni tra numeri interi attraverso una serie di addizioni. Se uno dei fattori (ad esempio **B**) puo' essere espresso nella forma $\mathbf{10^n}$, abbiamo visto che il prodotto di **A** per **B** consiste nell'aggiungere **n** zeri alla destra di **A**, cioe' nel far scorrere le cifre di **A** di **n** posti verso sinistra.
In relazione alla divisione, dati due numeri interi **A** e **B** positivi o negativi (con **B** diverso da zero), il quoziente tra **A** e **B** in valore assoluto e' pari a quante volte **B** e' interamente contenuto in **A** (sottrazioni successive); il resto della divisione in valore assoluto e' la parte di **A** che rimane dopo l'ultima sottrazione. Nel caso ad esempio di **A=350** e **B=100**, possiamo scrivere:
`1) 350 - 100 = 250, 2) 250 - 100 = 150, 3) 150 - 100 = 50, Q = 3, R = 50`
In sostanza **B=100** puo' essere sottratto **3** volte da **A**, per cui **Q=3**; alla fine rimane **50** che rappresenta il resto **R**. Se uno o entrambi i numeri **A** e **B** sono negativi, la **CPU** puo' seguire il metodo gia' illustrato in precedenza per la divisione tra numeri interi con segno; la nostra **CPU** puo' quindi effettuare anche divisioni tra numeri interi attraverso una serie di sottrazioni che a loro volta possono essere convertite in addizioni. Se il divisore **B** puo' essere espresso nella forma $\mathbf{10^n}$, abbiamo visto che il quoziente tra **A** e **B** consiste nel togliere **n** cifre dalla destra di **A**, cioe' nel far scorrere le cifre di **A** di **n** posti verso destra; le **n** cifre che escono da destra rappresentano il resto della divisione. Se **A** e' un intero positivo, nello scorrimento delle sue cifre verso destra bisogna riempire con zeri i posti rimasti liberi a sinistra (estensione del segno); se **A** e' un intero negativo, nello scorrimento delle sue cifre verso destra bisogna riempire con dei **9** i posti rimasti liberi a sinistra (estensione del segno).

La possibilita' di convertire in addizioni le sottrazioni, le moltiplicazioni e le divisioni, comporta enormi semplificazioni circuitali per le **CPU**; questo e' proprio quello che accade nella realta'. Infatti, i circuiti elettronici che devono effettuare moltiplicazioni e divisioni, eseguono in realta' una serie di somme, di negazioni e di scorrimenti di cifre.

3.6 Numeri reali

Da quanto abbiamo visto in questo capitolo, la **CPU** e' in grado di gestire via hardware solo ed esclusivamente numeri interi con o senza segno; molto spesso pero' si ha la necessita' di eseguire dei calcoli piuttosto precisi su numeri con la virgola (numeri reali). In questo caso si presentano due possibilita': o si scrive un apposito software capace di permettere alla **CPU** di maneggiare i numeri reali, oppure si ricorre al cosiddetto **coprocessore matematico**.
La gestione via software dei numeri reali e soprattutto la gestione delle operazioni che si possono eseguire sui numeri reali, comporta l'utilizzo di algoritmi piuttosto complessi; questa complessita' si ripercuote negativamente sulle prestazioni della **CPU**.

Fortunatamente, tutte le **CPU 80486DX** e superiori, contengono al loro interno anche il coprocessore matematico; questo dispositivo viene anche chiamato **FPU** o **Fast Processing Unit** (unita' di elaborazione veloce). La **FPU** permette di operare via hardware sui numeri reali, e mette a disposizione anche una serie di complesse funzioni matematiche gestibili ad altissima velocita'; tra queste funzioni si possono citare quelle logaritmiche, esponenziali, trigonometriche, etc. Gli algoritmi che consentono di implementare queste funzioni, si basano principalmente sugli sviluppi in serie di **Taylor** e **Mc Laurin**; come si sa' dalla matematica, attraverso gli sviluppi in serie e' possibile approssimare una funzione (anche trascendente), con un polinomio di grado prestabilito. La **FPU** viene trattata in un apposito capitolo della sezione **Assembly Avanzato**.

Giunti alla fine di questo capitolo, possiamo dire di aver appurato che sul computer qualsiasi informazione viene gestita sotto forma di numeri; abbiamo anche visto come la **CPU** rappresenta questi numeri e come vengono applicate ad essi le quattro operazioni matematiche fondamentali. Non abbiamo pero' ancora risposto ad una domanda: come fa' il computer a sapere che cosa e' un numero?
Non e' ancora tempo per dare una risposta perche' nel prossimo capitolo procederemo ad effettuare ulteriori (e colossali) semplificazioni relative sempre alla codifica numerica dell'informazione sul computer.

Capitolo 4 - Il sistema di numerazione binario

Nei precedenti capitoli abbiamo appurato che qualsiasi informazione da elaborare attraverso il computer, deve essere codificata in forma numerica; abbiamo anche visto che i codici numerici usati dal computer, sono costituiti da numeri interi basati sul sistema di numerazione posizionale. Nel Capitolo 3 in particolare, e' stato descritto il funzionamento di un computer immaginario che lavora con un sistema di numerazione posizionale in base **b=10**, costituito quindi dal seguente insieme di **10** simboli:

```
S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Questo computer immaginario ha solamente uno scopo didattico in quanto il suo modo di funzionare appare semplicissimo da capire; se provassimo pero' a realizzare veramente un computer di questo genere, andremmo incontro a notevoli difficolta' legate in particolare ad enormi complicazioni circuitali. Un computer "reale", per essere capace di lavorare con un sistema di numerazione in base **b=10**, dovrebbe avere dei circuiti interni capaci di riconoscere ben **10** simboli differenti; in termini pratici cio' si ottiene attraverso un segnale elettrico capace di assumere **10** diversi livelli di tensione. Come vedremo nel prossimo capitolo, una situazione di questo genere creerebbe anche una serie di problemi di carattere tecnico; possiamo dire quindi che l'idea di realizzare in pratica un computer capace di lavorare in base **b=10** e' assolutamente da scartare.
Il problema appena descritto puo' essere risolto in modo piuttosto efficace sfruttando una delle tante potentissime caratteristiche dei sistemi di numerazione posizionali; nel nostro caso, la caratteristica che possiamo sfruttare consiste nel fatto che nei sistemi di numerazione posizionali, il numero di simboli (**b**) da utilizzare e' del tutto arbitrario. E' chiaro che noi esseri umani troviamo naturale contare in base **b=10** perche' cosi' ci e' stato insegnato sin da bambini; per questo motivo, l'idea di metterci a contare utilizzando ad esempio una base **b=6** ci appare abbastanza assurda. In realta', non ci vuole molto a capire che tutti i concetti sulla matematica del computer esposti nel precedente capitolo, si applicano ad un sistema di numerazione posizionale in base **b** qualunque.

4.1 Sistema di numerazione posizionale in base b = 6

Per dimostrare che le proprieta' matematiche di un sistema di numerazione posizionale sono indipendenti dalla base **b** utilizzata, analizziamo quello che succede nel caso ad esempio di **b=6**; in base **b=6** ci troviamo ad operare con il seguente insieme di **6** simboli:

```
S = {0, 1, 2, 3, 4, 5}
```

Si tratta dei soliti simboli utilizzati nel mondo occidentale per la rappresentazione delle cifre; e' importante pero' tenere presente che questi simboli possono essere scelti in modo arbitrario. A titolo di curiosita', la Figura 1 mostra i simboli utilizzati per rappresentare le cifre nelle diverse aree linguistiche del mondo; per la corretta visualizzazione di questa tabella in ambiente **Windows**, bisogna disporre dei font **Lucida Sans Unicode** o **Arial Unicode MS**.

Figura 1

Lingua	Cifre
Occidentale	0 1 2 3 4 5 6 7 8 9
Arabo	٠ ١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩
Bengali	০ ১ ২ ৩ ৪ ৫ ৬ ৭ ৮ ৯
Oriya	୦ ୧ ୨ ୩ ୪ ୫ ୬ ୭ ୮ ୯
Malayalam	൦ ൧ ൨ ൩ ൪ ൫ ൬ ൭ ൮ ൯

Se ora proviamo a contare in base **b=6**, arrivati a **5** ci troviamo subito in difficolta'; se pensiamo pero' ai concetti esposti nel Capitolo 2, ci rendiamo subito conto che indipendentemente dagli aspetti formali, la situazione e' sostanzialmente identica al caso della base **b=10**.
Anche nel caso della base **6** quindi, i sei simboli usati rappresentano le cosiddette **unita'** (zero unita', una unita', due unita', ... , cinque unita'); aggiungendo una unita' a cinque unita' non sappiamo come rappresentare questo numero perche' non ci sono piu' simboli disponibili. Passiamo allora ai numeri a due cifre scrivendo **10**; in base **10** questo numero rappresenta zero unita' piu' un gruppo di dieci unita', mentre in base **6** rappresenta zero unita' piu' un gruppo di sei unita'.
Con i numeri a due cifre possiamo arrivare sino a **55** (cinque unita' piu' cinque gruppi di sei unita'); aggiungendo una unita' a **55** unita' ci imbattiamo nella situazione precedente. Introduciamo allora i numeri a tre cifre scrivendo **100**; in base **10** questo numero rappresenta zero unita' piu' zero gruppi di dieci unita' piu' un gruppo di cento unita' (dieci per dieci), mentre in base **6** rappresenta zero unita' piu' zero gruppi di sei unita' piu' un gruppo di trentasei unita' (sei per sei).
Con i numeri a tre cifre possiamo arrivare sino a **555** (cinque unita' piu' cinque gruppi di sei unita' piu' cinque gruppi di trentasei unita'); aggiungendo una unita' a **555** unita' ci imbattiamo ancora nella situazione precedente. Introduciamo allora i numeri a quattro cifre scrivendo **1000**; in base **10** questo numero rappresenta zero unita' piu' zero gruppi di dieci unita' piu' zero gruppi di cento unita' piu' un gruppo di mille unita' (dieci per dieci per dieci), mentre in base **6** rappresenta zero unita' piu' zero gruppi di sei unita' piu' zero gruppi di trentasei unita' piu' un gruppo di duecentosedici unita' (sei per sei per sei).
Come si puo' notare, si tratta dello stesso identico meccanismo utilizzato per contare in base **10**; si tenga presente che in basi diverse da **10**, i numeri vengono letti scandendo le loro cifre una per una da sinistra a destra (in base **6** ad esempio il numero **300** non deve essere letto come "trecento" ma come "tre zero zero").
Riprendiamo ora un esempio del Capitolo 2 e consideriamo il numero **333** in base **b=6**; il numero **333** e' formato da tre cifre uguali ma il loro peso e' differente. Osserviamo infatti che il **3** di destra rappresenta tre unita', il **3** centrale rappresenta tre gruppi di sei unita', mentre il **3** di sinistra rappresenta tre gruppi di trentasei unita'; possiamo scrivere allora:

```
333 = 300 + 30 + 3 = (3 * 100) + (3 * 10) + (3 * 1)

Tenendo presente che (in base 6):
```

$$
\begin{aligned}
1 &= 6^0 \\
10 &= 6^1 \\
100 &= 6^2
\end{aligned}
$$

```
possiamo scrivere:
```

$$
\begin{aligned}
&333 = 300 + 30 + 3 = \\
&(3 * 100) + (3 * 10) + (3 * 1) = \\
&(3 * 6^2) + (3 * 6^1) + (3 * 6^0)
\end{aligned}
$$

Come gia' sappiamo, dato un numero **n** in base **b** formato da **i** cifre aventi posizione:
`p = 0, 1, 2, ... i - 1` (a partire dalla cifra piu' a destra), si definisce peso di una cifra di **n** la potenza di ordine **p** della base **b**; dato allora il numero **333** in base **b=6**:

***** il peso del **3** piu' a destra e' $\mathbf{6^0 = 1}$;
***** il peso del **3** centrale e' $\mathbf{6^1 = 10}$;
***** il peso del **3** piu' a sinistra e' $\mathbf{6^2 = 100}$.

Naturalmente bisogna ricordarsi che stiamo operando in base **b=6**, per cui ad esempio il numero **100** rappresenta zero unita' piu' zero gruppi di sei unita' piu' un gruppo di trentasei unita'; questo

numero quindi va' letto "uno zero zero". Convertendo **100** in base **10** otteniamo:
$1 * 6^2 + 0 * 6^1 + 0 * 6^0 = 36$

A questo punto possiamo anche effettuare operazioni matematiche sui numeri in base **6**; a tale proposito ci puo' essere utile la tabellina delle addizioni in base **6** mostrata in Figura 2:

Figura 2

+	**0**	**1**	**2**	**3**	**4**	**5**
0	0	1	2	3	4	5
1	1	2	3	4	5	10
2	2	3	4	5	10	11
3	3	4	5	10	11	12
4	4	5	10	11	12	13
5	5	10	11	12	13	14

Con l'ausilio di questa tabella proviamo ora ad effettuare la seguente addizione:

```
132 +
325 =
-----
```

Partendo allora dalla colonna delle unita', in base alla tabella di Figura 2 si ottiene:
`2 unita' + 5 unita' = 11 unita'`
cioe' **1** unita' con riporto di un gruppo di **6** unita' che verra' aggiunto alla seconda colonna (che e' la colonna dei gruppi di **6** unita').
Passando alla seconda colonna otteniamo:
`3 gruppi di 6 unita' + 2 gruppi di 6 unita' = 5 gruppi di 6 unita'`
piu' un gruppo di **6** unita' di riporto = **10** gruppi di **6** unita', cioe' **0** gruppi di **6** unita' con riporto di un gruppo di **36** unita' che verra' aggiunto alla terza colonna (che e' la colonna dei gruppi di **36** unita').
Passando infine alla terza colonna otteniamo:
`1 gruppo di 36 unita' + 3 gruppi di 36 unita' = 4 gruppi di 36 unita'`
piu' un gruppo di **36** unita' di riporto = **5** gruppi di **36** unita'; il risultato finale sara' quindi **501**.
Come verifica osserviamo che in base **10** il primo addendo **132** diventa:
$1 * 6^2 + 3 * 6^1 + 2 * 6^0 = 36 + 18 + 2 = 56$
Il secondo addendo **325** diventa:
$3 * 6^2 + 2 * 6^1 + 5 * 6^0 = 108 + 12 + 5 = 125$
La somma **501** diventa:
$5 * 6^2 + 0 * 6^1 + 1 * 6^0 = 180 + 0 + 1 = 181$
Ripetendo la somma in base **10** otteniamo proprio:
`56 + 125 = 181`
Tutto cio' dimostra che le differenze tra i numeri espressi in basi diverse, sono solamente di carattere formale; la sostanza invece e' sempre la stessa. In definitiva, tutto cio' che viene fatto in base **10** puo' essere fatto in qualsiasi altra base ottenendo risultati perfettamente equivalenti.

4.2 Sistema di numerazione posizionale in base b = 2

Abbiamo appurato quindi che le proprieta' matematiche dei sistemi di numerazione posizionali, sono assolutamente indipendenti dalla base **b** che si vuole utilizzare; possiamo sfruttare allora questa caratteristica per realizzare un computer con la massima semplificazione circuitale possibile.
Come e' stato detto in precedenza, i diversi simboli che compongono un sistema di numerazione posizionale, vengono gestiti dai circuiti del computer sotto forma di diversi livelli di tensione assunti da un segnale elettrico; semplificare al massimo i circuiti del computer significa ridurre al minimo il numero di simboli da gestire, e cio' si ottiene attraverso l'adozione di un sistema di numerazione posizionale caratterizzato dalla minima base **b** possibile.
Scartando i casi come **b=0** e **b=1** che non hanno alcun senso, non ci resta che fissare la nostra attenzione sul sistema di numerazione posizionale in base **b=2**; in questo caso ci troviamo ad operare con il seguente insieme formato da due soli simboli:
`S = {0, 1}`
Per indicare il fatto che questo sistema di numerazione e' basato su due soli simboli, si parla anche di sistema **binario**; ciascuna cifra (**0** o **1**) del sistema binario viene chiamata **bit** (contrazione di **binary digit** = cifra binaria). Possiamo dire quindi che il bit rappresenta l'**unita' di informazione**, cioe' la quantita' minima di informazione gestibile dal computer.

Proviamo ora a contare in base **2** applicando le regole che gia' conosciamo; anche in questo caso quindi i due simboli usati rappresentano le cosiddette **unita'** (zero unita', una unita'). Aggiungendo una unita' a una unita' non sappiamo come rappresentare questo numero perche' non ci sono piu' simboli disponibili; passiamo allora ai numeri a due cifre scrivendo **10**, cioe' zero unita' piu' un gruppo di due unita'.
Con i numeri a due cifre possiamo arrivare sino a **11** (una unita' piu' un gruppo di due unita'); aggiungendo una unita' a **11** unita' ci imbattiamo nella situazione precedente. Introduciamo allora i numeri a tre cifre scrivendo **100**, cioe' zero unita' piu' zero gruppi di due unita' piu' un gruppo di quattro unita' (due per due).
Con i numeri a tre cifre possiamo arrivare sino a **111** (una unita' piu' un gruppo di due unita' piu' un gruppo di quattro unita'); aggiungendo una unita' a **111** unita' ci imbattiamo ancora nella situazione precedente. Introduciamo allora i numeri a quattro cifre scrivendo **1000**, cioe' zero unita' piu' zero gruppi di due unita' piu' zero gruppi di quattro unita' piu' un gruppo di otto unita' (due per due per due).
Si puo' constatare che piu' e' piccola la base **b**, piu' aumenta il numero di cifre necessarie per rappresentare uno stesso numero; ad esempio, il numero **318** espresso in base **10** con tre sole cifre, diventa **1250** in base **6** (quattro cifre) e **100111110** in base **2** (nove cifre). Non c'e' dubbio quindi che un essere umano incontrerebbe notevoli difficolta' nell'uso del sistema binario; nel caso del computer invece, i due soli simboli **0** e **1** vengono gestiti elettronicamente a velocita' vertiginose.
Consideriamo ora il numero **1111** in base **b=2**; questo numero e' formato da quattro cifre uguali che hanno pero' peso differente. Applicando infatti le proprieta' che gia' conosciamo sui sistemi di numerazione posizionali, possiamo scrivere:

```
1111 = 1000 + 100 + 10 + 1 =
(1 * 1000) + (1 * 100) + (1 * 10) + (1 * 1)

Tenendo presente che (in base 2):
```

$$
\begin{aligned}
1 &= 2^0 \\
10 &= 2^1 \\
100 &= 2^2 \\
1000 &= 2^3
\end{aligned}
$$

`possiamo scrivere:`

```
1111 = 1000 + 100 + 10 + 1 =
(1 * 1000) + (1 * 100) + (1 * 10) + (1 * 1) =
```
$(1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)$

Segue quindi che:

***** l'**1** piu' a destra ha posizione **0** e peso $\mathbf{2^0 = 1}$;
***** il secondo **1** ha posizione **1** e peso $\mathbf{2^1 = 10}$;
***** il terzo **1** ha posizione **2** e peso $\mathbf{2^2 = 100}$;
***** l'**1** piu' a sinistra ha posizione **3** e peso $\mathbf{2^3 = 1000}$.

In base **b=2** un numero come **1000** rappresenta zero unita' piu' zero gruppi di due unita' piu' zero gruppi di quattro unita' piu' un gruppo di otto unita'; questo numero quindi va' letto "uno zero zero zero". Convertendo **1000** in base **10** otteniamo:
$1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0 = 8$
La cifra piu' a destra di un numero binario e' quella di peso minore e viene chiamata **LSB** o **least significant bit** (bit meno significativo); la cifra piu' a sinistra di un numero binario e' quella di peso maggiore e viene chiamata **MSB** o **most significant bit** (bit piu' significativo).

Una volta introdotto il sistema di numerazione binario, e' facile capire a cosa si riferisce la definizione di **CPU** con architettura a **8** bit, a **16** bit, a **32** bit etc; si tratta ovviamente del numero massimo di cifre binarie che la **CPU** puo' gestire in un colpo solo (via hardware). Tra i vari esempi reali si possono citare le **CPU 8080** a **8** bit, l'**8086** a **16** bit, l'**80386/80486** a **32** bit e le **CPU** di classe **Pentium** a **64** bit.
Una **CPU** con architettura a **8** bit e' in grado di gestire via hardware solo numeri formati al massimo da **8** cifre binarie; se vogliamo lavorare invece con numeri formati da piu' di **8** cifre, dobbiamo procedere come al solito via software suddividendo i numeri stessi in gruppi di **8** cifre.
Una **CPU** con architettura a **16** bit e' in grado di gestire via hardware solo numeri formati al massimo da **16** cifre binarie; se vogliamo lavorare invece con numeri formati da piu' di **16** cifre, dobbiamo procedere via software suddividendo i numeri stessi in gruppi di **8** o **16** cifre.
Una **CPU** con architettura a **32** bit e' in grado di gestire via hardware solo numeri formati al massimo da **32** cifre binarie; se vogliamo lavorare invece con numeri formati da piu' di **32** cifre, dobbiamo procedere via software suddividendo i numeri stessi in gruppi di **8**, **16** o **32** cifre.

Come mai si usano proprio questi strani numeri come **8**, **16**, **32**, **64** etc?
Il perche' di questa scelta e' legato al fatto che stiamo lavorando in base **2** e, come vedremo anche in seguito, utilizzando in base **2** numeri che si possono esprimere sotto forma di potenza intera del due, si ottengono enormi vantaggi sia in termini di hardware che in termini di software. Per il momento possiamo limitarci ad osservare che:
$8 = 2^3$, $16 = 2^4$, $32 = 2^5$, $64 = 2^6$, etc.
Possiamo dire allora che nel mondo del computer il numero perfetto non e' il **3** ma il **2**!

A questo punto siamo finalmente in grado di convertire in base **2** tutte le considerazioni esposte nel precedente capitolo relativamente al nostro computer immaginario che lavora in base **10**; in questo modo abbiamo la possibilita' di constatare che l'adozione del sistema di numerazione posizionale in base **2**, comporta enormi semplificazioni nei calcoli matematici, che si traducono naturalmente nella semplificazione dei circuiti che eseguono i calcoli stessi.
Prima di tutto e' necessario introdurre alcune convenzioni utilizzate nel mondo dei computers; la prima cosa da dire riguarda il fatto che per velocizzare le operazioni, i computers gestiscono le informazioni sotto forma di gruppi di bit. Come al solito, il numero di bit contenuti in ogni gruppo

non viene scelto a caso, ma e' sempre una potenza intera del **2**; la Figura 3 mostra le convenzioni adottate sulle piattaforme hardware basate sulle **CPU** della famiglia **80x86**:

Figura 3 - Multipli del bit

```
 4 bit =  1 nibble
 8 bit =  2 nibble =  1 byte
16 bit =  4 nibble =  2 byte = 1 word
32 bit =  8 nibble =  4 byte = 2 word = 1 doubleword
64 bit = 16 nibble =  8 byte = 4 word = 2 doubleword = 1 quadword
```

La Figura 4 illustra altre convenzioni legate ai multipli del byte:

Figura 4 - Multipli del byte

1 kilobyte = 1 Kb = 2^{10} byte =   1024 byte
1 megabyte = 1 Mb = 2^{20} byte =   1048576 byte
1 gigabyte = 1 Gb = 2^{30} byte = 1073741824 byte

```
di conseguenza:

1 Kb = 1024 byte
1 Mb = 1024 Kb = (1024 * 1024) byte
1 Gb = 1024 Mb = (1024 * 1024) Kb = (1024 * 1204 * 1024) byte
```

Notiamo subito che, mentre in campo scientifico i prefissi **kilo**, **mega** e **giga** moltiplicano l'unita' di misura a cui vengono applicati, rispettivamente per mille, per un milione e per un miliardo, in campo informatico il discorso e' completamente diverso; infatti questa volta **kilo** indica la potenza intera del **2** piu' vicina a mille (**2^{10}**), **mega** indica la potenza intera del **2** piu' vicina ad un milione (**2^{20}**) e **giga** indica la potenza intera del **2** piu' vicina ad un miliardo (**2^{30}**).

4.3 Numeri interi senza segno

Cominciamo allora a lavorare con i numeri binari riferendoci per semplicita' ad un computer con architettura a **8** bit; tutti i concetti che verranno esposti in relazione ai numeri binari a **8** bit, si possono estendere facilmente ai numeri binari formati da un numero qualsiasi di bit. Da questo momento in poi, per evitare di fare confusione con i numeri espressi in base **10**, tutti i numeri binari verranno rappresentati con il suffisso **'b'**; questa e' la stessa convenzione adottata dal linguaggio **Assembly**.

Questa volta il nostro contachilometri e' formato da **8** cifre e conta in base **2**; partendo allora da **00000000b** e procedendo in marcia avanti, vedremo comparire in sequenza i numeri binari **00000000b**, **00000001b**, **00000010b**, **00000011b**, **00000100b** e cosi' via sino a **11111111b** che in base **10** rappresenta il numero **255** (**2^8-1**). Se ora percorriamo un'altro Km, il contachilometri ricomincera' da **00000000b**; come ormai gia' sappiamo, questo accade perche' il contachilometri lavora come al solito in modulo **b^n**. Nel nostro caso **b=2** e **n=8**, per cui il modulo che stiamo utilizzando e' **2^8 = 256** che in binario si rappresenta come **100000000b**; con **8** bit complessivamente possiamo rappresentare **256** codici numerici differenti attraverso i quali possiamo codificare i primi **256** numeri dell'insieme **N** ottenendo le corrispondenze mostrate in Figura 5:

Figura 5

```
00000000b rappresenta il valore    0
00000001b rappresenta il valore   +1
00000010b rappresenta il valore   +2
00000011b rappresenta il valore   +3
```

```
..................................
11111100b rappresenta il valore +252
11111101b rappresenta il valore +253
11111110b rappresenta il valore +254
11111111b rappresenta il valore +255
```

Attraverso questo sistema di codifica, la nostra **CPU** puo' gestire via hardware un sottoinsieme finito di **N** formato quindi dai seguenti **256** numeri naturali:
`0, 1, 2, 3, ..., 252, 253, 254, 255`

4.3.1 Addizione tra numeri interi senza segno

Vediamo innanzi tutto in Figura 6 la tabellina per le addizioni binarie tra singoli bit:

Figura 6

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (CF = 1)
```

Come si puo' notare, l'unico caso degno di nota e' l'ultimo dove sommando **1+1** si ottiene **10**, cioe' **0** con riporto di **1**; e' del tutto superfluo ricordare che in qualsiasi base il riporto massimo (per le addizioni) e il prestito massimo (per le sottrazioni) non puo' superare **1**.
La nostra **CPU** puo' sommare via hardware numeri binari a **8** bit; il valore massimo senza segno ottenibile con **8** bit e' **255**, per cui la **CPU** porra' **CF=1** solo se la somma tra due numeri interi senza segno fornisce un risultato superiore a **255** (**11111111b**).
In Figura 7 vediamo alcuni esempi che a questo punto dovrebbero essere ormai superflui perche' anche un bambino sarebbe in grado di effettuare questi calcoli con carta e penna:

Figura 7

```
1) 10011111b + 00111111b = 11011110b (CF = 0)
2) 11001100b + 10000000b = 01001100b (CF = 1)
3) 10101010b + 01010101b = 11111111b (CF = 0)
```

Se alla fine si ottiene **CF=0**, vuol dire che il risultato e' valido cosi' com'e'; in caso contrario dobbiamo aggiungere un **1** alla sinistra del risultato ottenendo quindi un valore a **9** cifre.
Il flag **CF** puo' essere utilizzato per sommare via software numeri interi formati da piu' di **8** bit; le cifre che formano questi numeri devono essere suddivise in gruppi. Come gia' sappiamo, per ottenere la massima efficienza possibile, conviene fare in modo che ciascun gruppo contenga un numero di cifre pari a quello che rappresenta l'architettura della **CPU** che si sta utilizzando; nel nostro caso, i numeri formati da piu' di **8** bit devono essere suddivisi in gruppi di **8** bit a partire da destra. La Figura 8 mostra un esempio pratico:

Figura 8

```
Vogliamo calcolare:

0111001011110000101 0b + 11000101001111001011b =
100111000001011010101b

Suddividiamo gli addendi in gruppi di 8 bit
ed eseguiamo l'addizione colonna per colonna:
```

```
00000111b 00101111b 00001010b +
00001100b 01010011b 11001011b =
-------------------------------

colonna 1: 00001010b + 11001011b = 11010101b      (CF = 0)
colonna 2: 00101111b + 01010011b + CF = 10000010b (CF = 0)
colonna 3: 00000111b + 00001100b + CF = 00010011b (CF = 0)

Unendo i tre risultati si ottiene:

000100111000001011010101b
```

L'ultima addizione ci da' **CF=0**, per cui il risultato e' valido cosi' com'e'; in presenza di un riporto finale (**CF=1**), avremmo dovuto aggiungere un **1** alla sinistra del risultato ottenendo quindi un numero a **25** cifre.

4.3.2 Sottrazione tra numeri interi senza segno

La Figura 9 mostra la tabellina per le sottrazioni binarie tra singoli bit:

Figura 9

```
0 - 0 = 0
0 - 1 = 1 (CF = 1)
1 - 0 = 1
1 - 1 = 0
```

Questa volta l'unico caso degno di nota e' il secondo dove la sottrazione **1-1** ci da' come risultato **1** con prestito di **1** dalle colonne successive; per giustificare questa situazione e' sufficiente pensare a quello che accade in qualsiasi base **b**. In generale, il prestito richiesto alle colonne successive dalla colonna in posizione **n** e' pari a $\mathbf{b^{n+1}}$; in base **b=2** quindi, se ci troviamo ad esempio nella prima colonna (**n=0**), chiederemo un prestito di $\mathbf{2^1=2}$ alle colonne successive, cioe' un prestito di **1** gruppo di **2** unita'. Ottenuto il prestito possiamo scrivere:
`2 - 1 = 10b - 1b = 1b`
Questa sottrazione indica che sottraendo **1** unita' da un gruppo di **2** unita', si ottiene ovviamente **1** unita'.
Se ci troviamo nella seconda colonna (**n=1**), chiederemo un prestito di $\mathbf{2^2=4}$ alle colonne successive, cioe' un prestito di **1** gruppo di **4** unita'. Ottenuto il prestito possiamo scrivere:
`4 - 2 = 100b - 10b = 10b`
Questa sottrazione indica che sottraendo **1** gruppo di **2** unita' da **1** un gruppo di **4** unita', si ottiene ovviamente **1** gruppo di **2** unita'.

La nostra **CPU** puo' sottrarre via hardware numeri binari a **8** bit; se il minuendo e' maggiore o uguale al sottraendo si ottiene un risultato positivo o nullo con **CF=0**. Se il minuendo e' minore del sottraendo si ottiene **CF=1**; questo prestito proviene come al solito dai successivi gruppi di **8** bit. La Figura 10 mostra alcuni esempi pratici:

Figura 10

```
1) 10011111B - 00111111B = 01100000B (CF = 0)
2) 10001100B - 11000000B = 11001100B (CF = 1)
3) 11111111b - 00001111b = 11110000b (CF = 0)
```

Se alla fine si ottiene **CF=0**, vuol dire che il risultato e' valido cosi' com'e'; in caso contrario la sottrazione si e' conclusa con la richiesta di un prestito ai successivi gruppi di **8** bit.

Il flag **CF** puo' essere utilizzato per sottrarre via software numeri interi formati da piu' di **8** bit; le cifre che formano questi numeri devono essere suddivise in gruppi. Nel nostro caso (**CPU** con architettura a **8** bit), ogni gruppo e' formato da **8** cifre; la Figura 11 mostra un esempio pratico:

Figura 11

```
Vogliamo calcolare:

101111011001111001100b - 001101111100111101101b = 100000110110110111111b

Suddividiamo minuendo e sottraendo in gruppi di 8 bit
ed eseguiamo la sottrazione colonna per colonna:

00001011b 10110011b 11001100b -
00000011b 01111100b 11101101b =
-------------------------------

colonna 1: 11001100b - 11101101b = 11011111b        (CF = 1)
colonna 2: (10110011b - CF) - 01111100b = 00110110b (CF = 0)
colonna 3: (00001011b - CF) - 00000011b = 00001000b (CF = 0)

Unendo i tre risultati si ottiene:

000010000011011011011111b
```

L'ultima sottrazione ci da' **CF=0**, per cui il risultato e' valido cosi' com'e'; in presenza di un prestito finale (**CF=1**), si ottiene un risultato negativo che non appartiene all'insieme dei numeri senza segno.

4.3.3 Moltiplicazione tra numeri interi senza segno

La Figura 12 mostra la tabellina per le moltiplicazioni binarie tra singoli bit:

Figura 12

```
0 x 0 = 0
0 x 1 = 0
1 x 0 = 0
1 x 1 = 1
```

Applicando i concetti esposti nel precedente capitolo, possiamo dire che moltiplicando tra loro due numeri binari a **8** bit, si ottiene come prodotto un numero binario che non puo' superare i **16** bit; la **CPU** moltiplica quindi i due fattori a **8** bit e ci restituisce il risultato a **16** bit disposto in due gruppi da **8** bit ciascuno. In Figura 13 vediamo un esempio pratico che simula lo stesso procedimento che si usa per eseguire la moltiplicazione con carta e penna; questo esempio dimostra chiaramente le notevoli semplificazioni che si ottengono eseguendo la moltiplicazione in base **2**:

Figura 13

```
11001010b x 10011100b =
-----------------------
        00000000b
       00000000b
      11001010b
     11001010b
    11001010b
   00000000b
  00000000b
 11001010b
-----------------------
0111101100011000b
```

Per verificare questo risultato osserviamo che in base **10** il primo fattore corrisponde a **202**, il secondo fattore corrisponde a **156**, mentre il prodotto corrisponde a **31512**; ricalcolando la moltiplicazione in base **10** otteniamo proprio:

```
202 x 156 = 31512
```

Analizzando la Figura 13 ci si rende subito conto che la **CPU** non ha bisogno di eseguire nessuna moltiplicazione; notiamo innanzi tutto la semplicita' con la quale si ottengono i prodotti parziali. Gli unici due casi che si possono presentare sono:

`11001010b x 0 = 00000000b` e `11001010b x 1 = 11001010b`

In generale, dato un fattore **A**, si possono presentare solo i seguenti due casi:

`A x 0 = 0` e `A x 1 = A`

Le cifre degli **8** prodotti parziali cosi' ottenuti vengono sottoposte a uno scorrimento verso sinistra; per il primo prodotto parziale lo scorrimento e' di zero posti, per il secondo e' di un posto, per il terzo e' di due posti e cosi' via. Alla fine i vari prodotti parziali vengono sommati tra loro in modo da ottenere il risultato a **16** bit.
Si tenga presente che queste sono solamente considerazioni di carattere teorico; in realta' le **CPU** utilizzano algoritmi molto piu' potenti ed efficienti, che spesso vengono tenuti segreti dalle case costruttrici.

Consideriamo infine il caso particolare che si presenta quando uno dei fattori puo' essere posto nella forma 2^n; dato ad esempio il numero binario **00001110b**, possiamo osservare che:

00001110b x 2^1 = 00001110b x 00000010b = 00011100b
00001110b x 2^2 = 00001110b x 00000100b = 00111000b
00001110b x 2^3 = 00001110b x 00001000b = 01110000b
....

In sostanza, moltiplicare un numero binario **A** (intero senza segno) per 2^n significa aggiungere **n** zeri alla destra di **A**; tutto cio' equivale a far scorrere di **n** posti verso sinistra tutte le cifre di **A**.
Questa proprieta' ci permette di velocizzare enormemente la moltiplicazione; osserviamo infatti che in questo caso la **CPU** evita di effettuare tutti i calcoli visibili in Figura 13.
Come al solito, un valore eccessivo di **n** puo' provocare il trabocco da sinistra di cifre significative del risultato; ad esempio, un qualunque numero binario a **8** bit, dopo **8** scorrimenti verso sinistra diventa in ogni caso **00000000b**!

4.3.4 Divisione tra numeri interi senza segno

La Figura 14 mostra la tabellina per le divisioni binarie tra singoli bit; chiaramente sono proibiti tutti casi per i quali il divisore e' **0**.

Figura 14
0 / 1 = 0
1 / 1 = 1

Applicando i concetti esposti nel precedente capitolo, possiamo dire che dividendo tra loro due numeri binari a **8** bit, si ottiene un quoziente non superiore al dividendo e un resto non superiore al divisore; la **CPU** esegue quindi la divisione intera e ci restituisce un quoziente a **8** bit e un resto a **8** bit.
In Figura 15 vediamo un esempio pratico che simula lo stesso procedimento che si usa per eseguire le divisioni con carta e penna:

Figura 15

```
11100110b / 00010000b =
10000b      -----------
```

```
---------   1110b
 11001b
 10000b
 --------
  10011b
  10000b
  -------
   00110b
   00000b
   ------
     110b
```

La divisione ci fornisce **Q=1110b** e **R=0110b**; in base **10** il dividendo diventa **230**, il divisore diventa **16**, il quoziente diventa **14** e il resto diventa **6**. Ricalcolando la divisione in base **10** otteniamo infatti:
`230 / 16 = Q = 14, R = 6`
Analizzando la Figura 15 ci si rende subito conto che la **CPU** non ha bisogno di eseguire nessuna divisione; tutto si svolge infatti attraverso confronti tra numeri binari, sottrazioni e scorrimenti di cifre.
Se il dividendo **A** e' minore del divisore **B**, la **CPU** ci restituisce direttamente **Q=0** e **R=A**; se il dividendo **A** e' uguale al divisore **B**, la **CPU** ci restituisce direttamente **Q=1** e **R=0**. Se il dividendo e' maggiore del divisore, viene applicato il metodo mostrato in Figura 15; come si puo' notare, si procede esattamente come per la base **10**.
Anche in questo caso pero' si tratta di considerazioni di carattere teorico; in realta' le **CPU** eseguono la divisione utilizzando algoritmi molto piu' efficienti. Nei capitoli successivi verra' mostrata un'altra tecnica che permette alla **CPU** di eseguire le divisioni molto piu' rapidamente; in ogni caso, appare chiaro il fatto che le divisioni e le moltiplicazioni vengono eseguite dalla **CPU** molto piu' lentamente rispetto alle addizioni e alle sottrazioni.

Consideriamo infine il caso particolare che si presenta quando il divisore puo' essere posto nella forma $\mathbf{2^n}$; dato ad esempio il numero binario **01110000b**, possiamo osservare che:

```
01110000b / 2^1 = 01110000b / 00000010b = 00111000b
01110000b / 2^2 = 01110000b / 00000100b = 00011100b
01110000b / 2^3 = 01110000b / 00001000b = 00001110b
....................................................
```

In sostanza, dividere un numero binario **A** (intero senza segno) per $\mathbf{2^n}$ significa aggiungere **n** zeri alla sinistra di **A**; tutto cio' equivale a far scorrere di **n** posti verso destra tutte le cifre di **A**. Questa proprieta' ci permette di velocizzare enormemente la divisione; osserviamo infatti che in questo caso la **CPU** evita di effettuare tutti i calcoli visibili in Figura 15.
Nella divisione da **A** per $\mathbf{2^n}$, le **n** cifre meno significative di **A** traboccano da destra; come abbiamo visto nel precedente capitolo, queste **n** cifre formano il resto della divisione.

4.4 Numeri interi con segno

Come gia' sappiamo, con **8** bit possiamo rappresentare $\mathbf{2^8=256}$ diversi codici numerici; in analogia a quanto e' stato fatto per la base **10**, suddividiamo questi **256** codici in due gruppi da **128**.
Osserviamo ora che con il nostro contachilometri binario a **8** cifre, partendo da **00000000b** e procedendo in marcia avanti, vedremo comparire la sequenza:
`00000000b, 00000001b, 00000010b, 00000011b, 00000100b, ...`
Partendo sempre da **00000000b** e procedendo in marcia indietro, vedremo comparire la sequenza:
`11111111b, 11111110b, 11111101b, 11111100b, 11111011b, ...`

Possiamo dire quindi che tutti i codici da **0** a **127** rappresentano i numeri interi positivi compresi tra **+0** e **+127**; tutti i codici da **128** a **255** rappresentano i numeri interi negativi compresi tra **-128** e **-1**.
Si ottengono in questo modo le corrispondenze mostrate in Figura 16:

Figura 16

```
01111111b rappresenta il valore +127
01111110b rappresenta il valore +126
....................................
00000011b rappresenta il valore   +3
00000010b rappresenta il valore   +2
00000001b rappresenta il valore   +1
00000000b rappresenta il valore   +0
11111111b rappresenta il valore   -1
11111110b rappresenta il valore   -2
11111101b rappresenta il valore   -3
....................................
10000001b rappresenta il valore -127
10000000b rappresenta il valore -128
```

La Figura 16 evidenzia una delle tante semplificazioni legate all'uso del sistema binario; osserviamo infatti che tutti i numeri interi positivi hanno il **MSB** che vale **0**, mentre tutti i numeri interi negativi hanno il **MSB** che vale **1**. Questa situazione permette alla **CPU** di individuare con estrema facilita' il segno di un numero.

4.4.1 Complemento a 1 e complemento a 2

Il nostro contachilometri binario a **8** cifre lavora in modulo **2^8=256** (che in binario si scrive **100000000b**); ogni volta che superiamo **11111111b**, il contachilometri ricomincia a contare da **00000000b**. Possiamo dire quindi che nell'aritmetica modulare del nostro contachilometri, **00000000b** equivale a **100000000b** (cioe' **0** equivale a **256**); possiamo scrivere quindi:

```
-1 = 0 - 1 = 256 - 1 = 100000000b - 00000001b = 11111111b
-2 = 0 - 2 = 256 - 2 = 100000000b - 00000010b = 11111110b
-3 = 0 - 3 = 256 - 3 = 100000000b - 00000011b = 11111101b
..........................................................
```

Con questa tecnica possiamo ricavare facilmente la codifica binaria di un numero intero negativo; dato un numero intero positivo **n** compreso tra **+1** e **+127**, la codifica binaria del corrispondente numero negativo (**-n**) si ottiene dalla formula:
`100000000b - n = 256 - n`
Nell'eseguire questa sottrazione con carta e penna, possiamo evitare il fastidio di eventuali prestiti osservando che:
`100000000b - n = (11111111b + 1b) - n = (11111111b - n) + 1b`
Vediamo un esempio pratico relativo al numero positivo **00000011b** (**+3**); per ricavare il corrispondente numero negativo **-3**, possiamo scrivere:
`(11111111b - 00000011b) + 1b = 11111100b + 1 = 11111101b`
Attraverso la Figura 16 possiamo constatare che **11111101b** e' proprio la codifica binaria del numero negativo **-3**.
Analizzando il calcolo appena effettuato, ci accorgiamo che la prima sottrazione non fa' altro che invertire tutti i bit del numero **n**; in sostanza, tutti i bit che valgono **0** diventano **1**, e tutti i bit che valgono **1** diventano **0**. L'operazione che consiste nell'inversione di tutti i bit di un numero binario, prende il nome di **complemento a 1**; per svolgere questa operazione, tutte le **CPU** della famiglia **80x86** forniscono una apposita istruzione chiamata **NOT**. La successiva somma di **NOT(n)** con **1b** ci fornisce l'opposto del numero **n**, cioe' il numero **n** cambiato di segno; nel complesso, l'operazione

che porta al cambiamento di segno di un numero binario **n** prende il nome di **complemento a 2**. Per svolgere questa operazione, tutte le **CPU** della famiglia **80x86** forniscono una apposita istruzione chiamata **NEG**; in base alle considerazioni appena esposte possiamo dire che:
`NEG(n) = NOT(n) + 1`
L'utilizzo del sistema binario permette quindi alla **CPU** di **negare** (cambiare di segno) facilmente un numero **n**; la **CPU** non deve fare altro che invertire tutti i bit di **n** e sommare **1** al risultato ottenuto.
Provando ad esempio a negare **11111101b** (**-3**) dovremmo ottenere:
`-(-3) = +3 = 00000011b`
Infatti:
`(11111111b - 11111101b) + 1b = 00000010b + 1b = 00000011b = +3`

In definitiva, con la tecnica appena vista la nostra **CPU** puo' gestire via hardware tutti i numeri interi con segno che formano il seguente sottoinsieme finito di **Z**:
`-128, -127, ..., -3, -2, -1, +0, +1, +2, ..., +126, +127`
Il sistema di codifica binaria di questi numeri prende il nome di `rappresentazione in complemento a` **2** `dei numeri interi con segno in modulo` **256**.

4.4.2 Addizione e sottrazione tra numeri interi con segno

Consideriamo innanzi tutto i due codici **00111100b** e **10111100b**; trattando questi codici come numeri interi senza segno, otteniamo:
`00111100b + 10111100b = (+60) + (+188) = +248 = 11111000b`
Trattando questi codici come numeri interi con segno, otteniamo:
`00111100b + 10111100b = (+60) + (-68) = -8 = 11111000b`
Si conferma quindi il fatto che nell'eseguire addizioni e sottrazioni la **CPU** non ha bisogno di dover distinguere tra numeri interi con o senza segno; avremo quindi un'unica istruzione (**ADD**) per le addizioni e un'unica istruzione (**SUB**) per le sottrazioni.

Se il programmatore ha bisogno di distinguere tra numeri interi con segno e numeri interi senza segno, deve servirsi come al solito dei flags **CF**, **OF** e **SF**; analizziamo innanzi tutto in Figura 17 una serie di esempi relativi a numeri binari a **8** bit:

Figura 17
```
1) 00110111b + 00011110b = 01010101b (CF = 0, SF = 0, OF = 0)
2) 11111010b + 11110000b = 11101010b (CF = 1, SF = 1, OF = 0)
3) 01111000b + 01101110b = 11100110b (CF = 0, SF = 1, OF = 1)
4) 10000010b + 10001100b = 00001110b (CF = 1, SF = 0, OF = 1)
```

Interpretando questi codici come numeri interi senza segno abbiamo:
`1) 55 + 30 = 85, 2) 250 + 240 = 234, 3) 120 + 110 = 230, 4) 130 + 140 = 14`
Negli esempi **1** e **3** si ha **CF=0**, per cui il risultato e' valido cosi' com'e'; negli esempi **2** e **4** si ha **CF=1** in quanto il risultato ottenuto supera **255**. In questo caso dobbiamo aggiungere un **1** alla sinistra del risultato binario; considerando l'esempio **2** abbiamo infatti:
`111101010b = 490 = 250 + 240`
Interpretando i codici numerici di Figura 17 come numeri interi con segno, abbiamo:
`1) (+55) + (+30) = +85, 2) (-6) + (-16) = -22, 3) (+120) + (+110) = -26, 4) (-126) + (-116) = +14`
Negli esempi **1** e **2** siamo rimasti nell'intervallo **[-128, + 127]** (**OF=0**), per cui il risultato e' valido cosi' com'e'; l'esempio **1** produce un risultato positivo (**SF=0**), mentre l'esempio **2** produce un risultato negativo (**SF=1**).
Negli esempi **3** e **4** si ha **OF=1**, in quanto il risultato e' andato in overflow; osserviamo infatti che nell'esempio **3** sommando due numeri positivi si ottiene un risultato "troppo positivo" che supera

+127 e sconfina nell'insieme dei numeri negativi a **8** bit. Analogamente, nell'esempio **4** sommando due numeri negativi si ottiene un risultato "troppo negativo" che scende sotto **-128** e sconfina nell'insieme dei numeri positivi a **8** bit.

Se abbiamo a che fare con addizioni e sottrazioni che coinvolgono numeri interi con segno formati da piu' di **8** bit, dobbiamo procedere via software utilizzando gli algoritmi che gia' conosciamo; come e' stato spiegato nel Capitolo 3, quando si opera con i numeri interi con segno bisogna stabilire in anticipo il modulo che vogliamo utilizzare, in modo da poter ottenere la corretta rappresentazione in complemento a **2** dei numeri stessi.
Supponiamo ad esempio di voler operare su numeri interi con segno a **32** bit; in questo caso il modulo e' pari a $\mathbf{2^{32}}$. In analogia a quanto abbiamo visto per i numeri interi con segno a **8** bit, i numeri interi positivi saranno rappresentati dalle codifiche comprese tra **00000000000000000000000000000000b** e **01111111111111111111111111111111b**, mentre i numeri interi negativi saranno rappresentati dalle codifiche comprese tra **10000000000000000000000000000000b** e **11111111111111111111111111111111b**; proviamo allora ad effettuare la seguente addizione:
`(+1029443343) + (-2000000000) = -970556657`
La codifica binaria di **+1029443343** e' **00111101010111000000111100001111b**, mentre la codifica binaria di **+2000000000** e' **01110111001101011001010000000000b**; prima di tutto dobbiamo cambiare di segno **+2000000000**. A tale proposito invertiamo tutti i bit di **+2000000000** e sommiamo **1** al risultato ottenendo:
`10001000110010100110101111111111b + 1b = 10001000110010100110110000000000b`
Come si puo' notare, il **MSB** vale **1** e ci indica quindi la codifica di un numero negativo; a questo punto possiamo procedere con l'addizione come viene mostrato in Figura 18:

Figura 18

```
Suddividiamo gli addendi in gruppi di 8 bit
ed eseguiamo l'addizione colonna per colonna:

00111101b 01011100b 00001111b 00001111b +
10001000b 11001010b 01101100b 00000000b =
-----------------------------------------

colonna 1: 00001111b + 00000000b = 00001111b      (CF = 0)
colonna 2: 00001111b + 01101100b + CF = 01111011b (CF = 0)
colonna 3: 01011100b + 11001010b + CF = 00100110b (CF = 1)
colonna 4: 00111101b + 10001000b + CF = 11000110b (CF = 0, OF = 0, SF = 1)

Unendo i tre risultati si ottiene:

11000110001001100111101100001111b
```

Le addizioni relative alle colonne **2**, **3** e **4** coinvolgono anche **CF** per tener conto di eventuali riporti provocati dalle precedenti addizioni; terminata l'ultima addizione (colonna **4**) dobbiamo consultare non **CF** ma **OF** per sapere se il risultato e' valido (**OF=0**) o se si e' verificato un overflow (**OF=1**).
Se il risultato e' valido, possiamo consultare **SF** per conoscerne il segno; i flags **OF** e **CF** vanno consultati solo alla fine perche' come sappiamo, il segno degli addendi si trova codificato nel loro **MSB**.
Se ora convertiamo in base **10** il risultato dell'addizione, otteniamo **3324410639**; per i numeri interi con segno a **32** bit questa e' proprio la codifica del numero negativo:
$-(2^{32} - 3324410639) = -970556657$

4.4.3 Moltiplicazione tra numeri interi con segno

Come abbiamo visto nel precedente capitolo, in presenza di una moltiplicazione tra numeri interi con segno, la **CPU** memorizza il segno del risultato, converte se necessario i due fattori in numeri positivi, esegue la moltiplicazione e infine applica il segno al risultato; moltiplicando due numeri interi con segno a **8** bit, si ottiene un risultato a **16** bit che verra' restituito dalla **CPU** in due gruppi da **8** bit.
Abbiamo anche visto che la moltiplicazione provoca un cambiamento di modulo, per cui la **CPU** ha bisogno di sapere se vogliamo operare sui numeri interi senza segno o sui numeri interi con segno; come gia' sappiamo, per gestire questi due casi dobbiamo utilizzare le due istruzioni **MUL** e **IMUL**.
Supponiamo ad esempio di voler eseguire la seguente moltiplicazione:
`250 x 120 = 30000`
Con la nostra **CPU** a **8** bit, il numero **250** viene codificato come **11111010b**, mentre il numero **120** viene codificato come **01111000b**; interpretando questi due codici come numeri interi senza segno, si ottiene:
`11111010b x 01111000b = 01110101b 00110000b`
Per i numeri interi con segno il codice **11111010b** rappresenta il numero negativo **-6**, per cui la moltiplicazione diventa in questo caso:
`(-6) x (+120) = -720`
La **CPU** nega quindi il primo numero ottenendo:
`NEG(11111010b) = 00000101b + 1b = 00000110b`
che e' proprio la codifica binaria di **+6**. La moltiplicazione binaria produce quindi:
`00000110b x 01111000b = 00000010b 11010000b`
Il risultato deve essere negativo, per cui la **CPU** lo nega in modulo $\mathbf{2^{16}}$ ottenendo:
`NEG(00000010b 11010000b) = 11111101b 00101111b + 1b = 11111101b 00110000b`
Per i numeri interi con segno a **16** bit questa e' proprio la codifica binaria di **-720**; infatti:
2^{16} `- 720 = 64816 = 11111101b 00110000b`
In definitiva, nell'insieme dei numeri interi senza segno si ottiene **01110101b 00110000b**, mentre nell'insieme dei numeri interi con segno si ottiene **11111101b 00110000b**; a causa del cambiamento di modulo, i due risultati sono completamente diversi tra loro.

Se uno dei due fattori puo' essere posto nella forma $\mathbf{2^n}$, possiamo velocizzare la moltiplicazione attraverso l'istruzione **SAL**; come gia' sappiamo, a causa del fatto che il segno di un numero e' codificato nel suo **MSB**, il risultato prodotto da **SAL** e' identico a quello prodotto da **SHL**.
Consideriamo ad esempio il numero **00111100b** e moltiplichiamolo per $\mathbf{2^4}$; applicando **SHL** otteniamo:
`SHL(00111100b, 4) = 11000000b`
Applicando **SAL** otteniamo:
`SAL(00111100b, 4) = 11000000b`
Le due istruzioni **SHL** e **SAL** sono quindi interscambiabili; le **CPU** della famiglia **80x86** le forniscono entrambe solo per motivi di simmetria con la coppia **SHR**, **SAR**.

4.4.4 Divisione tra numeri interi con segno

Anche per la divisione tra numeri interi con segno, abbiamo visto che la **CPU** converte se necessario il dividendo e il divisore in numeri interi positivi, esegue la divisione intera e applica infine il segno al quoziente e al resto; questa operazione produce un cambiamento di modulo, per cui la **CPU** ha bisogno di sapere se vogliamo operare sui numeri interi senza segno o su quelli con segno. Per poter gestire questi due casi dobbiamo utilizzare le due apposite istruzioni **DIV** e **IDIV**.
Consideriamo ad esempio i due numeri **250** e **120** che vengono codificati in binario come **11111010b** e **01111000b**; trattando questi codici come numeri interi senza segno, si ottiene:
`250 / 120 = Q = 2, R = 10`
In binario si ha quindi:

```
11111010b / 01111000b = Q = 00000010b, R = 00001010b
```

Trattando invece i due codici come numeri interi con segno, si ottiene:

```
(-6) / (+120) = Q = 0, R = -6
```

In binario la **CPU** nota che **11111010b** e' un numero negativo, per cui lo nega ottenendo:

```
NEG(11111010b) = 00000101b + 1b = 00000110b
```

che e' proprio la rappresentazione di **+6**; a questo punto viene eseguita la divisione tra numeri positivi:

```
00000110b / 01111000b = Q = 00000000, R = 00000110b
```

Siccome il resto deve essere negativo, la **CPU** lo nega ottenendo:

```
NEG(00000110b) = 11111001b + 1b = 11111010b
```

che e' proprio la rappresentazione a **8** bit di **-6**.
In definitiva, nell'insieme dei numeri interi senza segno otteniamo **Q=00000010b** e **R=00001010b**; nell'insieme dei numeri interi con segno otteniamo invece **Q=00000000b** e **R=11111010b**; a causa del cambiamento di modulo, i due risultati sono completamente diversi tra loro.

Se il divisore puo' essere posto nella forma $\mathbf{2^n}$, possiamo velocizzare la divisione attraverso l'istruzione **SAR**; questa istruzione e' completamente diversa da **SHR** in quanto fa' scorrere verso destra le cifre del dividendo preservando il **MSB** del dividendo stesso.
Consideriamo ad esempio il numero **00111101b** e dividiamolo per $\mathbf{2^2}$; applicando **SHR** otteniamo:

```
SHR(00111101b, 2) = 00001111b
```

Applicando **SAR** otteniamo:

```
SAR(00111101b, 2) = 00001111b
```

I due risultati sono identici in quanto **00111101b** e' positivo anche nell'insieme dei numeri interi con segno.
Consideriamo ora il numero **10110001b** e dividiamolo per $\mathbf{2^2}$; applicando **SHR** otteniamo:

```
SHR(10110001b, 2) = 00101100b
```

Applicando **SAR** otteniamo:

```
SAR(10110001b, 2) = 11101100b
```

Questa volta i due risultati sono differenti in quanto **10110001b** nell'insieme dei numeri con segno e' negativo; l'istruzione **SHR** aggiunge zeri alla sinistra del numero, mentre **SAR** tiene conto del segno e aggiunge degli **1** alla sinistra del numero.
Come e' stato detto nel precedente capitolo, in alcuni casi **SAR** produce un risultato sbagliato; il perche' viene spiegato in dettaglio in un altro capitolo.

4.5 Estensione del segno

Applichiamo ora alla base **b=2** i concetti esposti nel precedente capitolo in relazione al cambiamento di modulo per i numeri interi con segno; vediamo ad esempio come si deve procedere per convertire numeri interi con segno a **8** bit in numeri interi con segno a **16** bit.
Per effettuare questa conversione non dobbiamo fare altro che applicare tutti i concetti relativi alla rappresentazione in complemento a **2** dei numeri interi con segno a **16** bit; con **16** bit possiamo rappresentare $\mathbf{2^{16}=65536}$ codici numerici (da **0** a **65535**). Tutti i codici da **0000000000000000b** a **0111111111111111b** rappresentano i numeri interi positivi compresi tra **+0** e **+32767**; tutti i codici da **1000000000000000b** a **1111111111111111b** rappresentano i numeri interi negativi compresi tra **-32768** e **-1**.
Si vede subito allora che ad esempio il numero positivo a **8** bit **00111100b** diventa **0000000000111100b**; analogamente, il numero positivo a **8** bit **01111111b** diventa **0000000001111111b**.
Il numero negativo a **8** bit **11000111b** (**-57**) diventa **1111111111000111b**; infatti:

2^{16} - 57 = 65479 = 1111111111000111b

In sostanza, per prolungare da **8** a **16** bit un numero intero positivo, non dobbiamo fare altro che

aggiungere **8** zeri alla sua sinistra; per prolungare da da **8** a **16** bit un numero intero negativo, non dobbiamo fare altro che aggiungere **8** uno alla sua sinistra.

4.6 Proprieta' generali delle codifiche numeriche a n cifre in base b

Una volta che abbiamo scelto la base **b** del sistema di numerazione posizionale su cui operare, e una volta stabilito il numero massimo di cifre disponibili, restano determinate in modo univoco tutte le caratteristiche dell'insieme numerico che dobbiamo codificare; in particolare, possiamo ricavare una serie di proprieta' generali relative ai limiti inferiore e superiore dei numeri rappresentabili.
Nel precedente capitolo abbiamo avuto a che fare con un computer immaginario che lavora con codici numerici a **5** cifre espressi in base **b=10**; la Figura 19 mostra le caratteristiche dei numeri interi rappresentabili con questa codifica:

Figura 19

```
Base b = 10
Numero cifre = 5
Numero codici rappresentabili = 100000 = 10^5

Numeri interi senza segno:

   Min. = 0
   Max. = 99999 = 10^5 - 1

Numeri interi con segno:

   Min. = -50000 = -(10^5 / 2)
   Max. = +49999 = +((10^5 / 2) - 1)
```

In questo capitolo abbiamo analizzato il caso di un computer reale che lavora con codici numerici a **8** cifre espressi in base **b=2**; la Figura 20 mostra le caratteristiche dei numeri interi rappresentabili con questa codifica:

Figura 20

```
Base b = 2
Numero cifre = 8
Numero codici rappresentabili = 256 = 2^8

Numeri interi senza segno:

   Min. = 0
   Max. = 255 = 2^8 - 1

Numeri interi con segno:

   Min. = -128 = -(2^8 / 2) = -2^7
   Max. = +127 = +((2^8 / 2) - 1) = +(2^7 - 1)
```

Analizzando la Figura 19 e la Figura 20 possiamo ricavare facilmente le proprieta' generali dei numeri rappresentabili con un sistema di codifica numerica a **n** cifre in base **b**; la Figura 21 mostra proprio queste proprieta':

Figura 21

```
Base b
Numero cifre = n
Numero codici rappresentabili = b^n

Numeri interi senza segno:

   Min. = 0
   Max. = b^n - 1

Numeri interi con segno:

   Min. = -(b^n / 2)
   Max. = +((b^n / 2) - 1)
```

Tutti i linguaggi di programmazione di alto livello forniscono al programmatore una serie di tipi di dati interi con o senza segno; questi dati vengono visualizzati sullo schermo in formato decale (base **10**), ma ovviamente vengono gestiti internamente in base **2**. Le formule illustrate nella Figura 21 ci permettono di capire facilmente il perche' delle caratteristiche di questi tipi di dati.
Consideriamo in particolare i tipi di dati interi forniti dai linguaggi di alto livello destinati alle **CPU** con architettura a **16** bit; nel caso ad esempio del linguaggio **C**, abbiamo a disposizione i tipi di dati interi mostrati in Figura 22:

Figura 22 - Tipi di dati interi del C

Interi senza segno			
Nome	**bit**	**Min.**	**Max.**
unsigned char	8	0	255
unsigned int	16	0	65535
unsigned long	32	0	4294967295
Interi con segno			
Nome	**bit**	**Min.**	**Max.**
signed char	8	-128	+127
signed int	16	-32768	+32767
signed long	32	-2147483648	+2147483647

La Figura 23 mostra i principali tipi di dati interi supportati dal linguaggio **Pascal**:

Figura 23 - Tipi di dati interi del Pascal

Interi senza segno			
Nome	**bit**	**Min.**	**Max.**
Byte	8	0	255
Word	16	0	65535
Interi con segno			
Nome	**bit**	**Min.**	**Max.**
Shortint	8	-128	+127
Integer	16	-32768	+32767

Figura 23 - Tipi di dati interi del Pascal

Interi senza segno			
Nome	**bit**	**Min.**	**Max.**
Longint	32	-2147483648	+2147483647

La Figura 24 infine mostra i due soli tipi di dati interi supportati dal linguaggio **BASIC**:

Figura 24 - Tipi di dati interi del BASIC

Interi con segno			
Nome	**bit**	**Min.**	**Max.**
INTEGER	16	-32768	+32767
LONG	32	-2147483648	+2147483647

Come si puo' notare, i compilatori e gli interpreti dei linguaggi di alto livello (cioe' i programmi che traducono il codice sorgente in codice macchina), supportano spesso tipi di dati interi aventi un numero di bit superiore a quello dell'architettura della **CPU**; questi tipi di dati vengono gestiti via software dai compilatori o dagli interpreti stessi.

Nel precedente capitolo e' stato citato il caso del gioco **Tetris** originariamente scritto in **BASIC**; il punteggio del gioco veniva gestito attraverso il tipo **INTEGER** (intero con segno a **16** bit), e i giocatori piu' bravi riuscivano facilmente a provocare un overflow. In sostanza, bastava superare i **32767** punti per veder comparire sullo schermo il punteggio negativo **-32768**; in base ai concetti appena esposti possiamo giustificare facilmente questa situazione. Osserviamo che in binario il valore a **16** bit **32767** si scrive **0111111111111111b**; abbiamo quindi:
0111111111111111b + 1b = 1000000000000000b
Ma per i numeri interi con segno a **16** bit questa e' proprio la rappresentazione in complemento a **2** del numero negativo **-32768**; infatti:
2^{16} - 32768 = 1000000000000000b
Chi avesse ancora dei dubbi puo' verificare in pratica questi concetti attraverso i programmi mostrati nella Figura 25; questi programmi fanno riferimento naturalmente a compilatori e interpreti destinati alla modalita' reale **8086** supportata dal **DOS**.

Figura 25 - Overflow di un intero con segno a 16 bit

```
/* versione C */

#include < stdio.h >

int main(void)
{
   signed short int  i = +32767;

   i++;
   printf("i = %d\n", i);

   return 0;
}
```

```
{ versione Pascal }

const i: Integer = +32767;

begin
   i:= i + 1;
   WriteLn('i = ', i);
end.
```

```
' versione BASIC

i% = +32767;

i% = i% + 1
PRINT "i% = "; i%
END
```

Nel caso degli interpreti **BASIC**, se si esegue il programma dall'interno dell'editor integrato, compare una finestra che informa dell'avvenuto overflow; se invece si ha a che fare con una versione compilata del **BASIC** (come il **Quick Basic**) e si effettua la compilazione dal prompt del **DOS**, l'overflow si verifichera' solo in fase di esecuzione del programma.

4.7 Rappresentazione dei numeri in altre basi

Come molti avranno notato, passando da base **10** a base **2** (cioe' da una base ad un'altra piu' piccola), e' necessario usare molte piu' cifre per rappresentare uno stesso numero; volendo rappresentare ad esempio il numero **3000000** (tre milioni) in base **2** si ottiene **001011011100011011000000b**. Volendo esprimere in base **2** numeri dell'ordine di qualche miliardo, si rischia di essere sepolti da una valanga di **0** e **1**. Il computer gestisce i numeri binari a velocita' vertiginose, mentre noi esseri umani incontriamo notevoli difficolta' ad usare i numeri in questo formato; d'altra parte per un programmatore sarebbe assurdo rinunciare ai notevoli vantaggi che si ottengono lavorando in base **2**. Capita frequentemente ad esempio di dover gestire nei programmi dati numerici dove ogni singolo bit ha un preciso significato; e' evidente allora che sarebbe meglio avere questi numeri espressi in base **2** piuttosto che in base **10** visto che in quest'ultimo caso risulta piuttosto difficile distinguere i singoli bit.
E' necessario allora trovare un formato di rappresentazione dei numeri binari, che presenti le seguenti caratteristiche fondamentali:

***** possibilita' di rappresentare i numeri binari in forma compatta;
***** estrema semplicita' di conversione da una base all'altra;
***** facilita' di individuazione del valore dei singoli bit.

Il primo requisito viene soddisfatto scegliendo ovviamente una base maggiore di **2**; il secondo e piu' importante requisito viene invece soddisfatto facendo in modo che la nuova base si possa esprimere sotto forma di potenza intera di **2** per i motivi che vedremo tra poco.
Si puo' subito notare che la base **10** soddisfa il primo requisito ma non il secondo perche' non e' possibile esprimere **10** come $\mathbf{2^n}$ con **n** intero positivo; per questo motivo, utilizzare nei programmi (non solo in **Assembly**) numeri in base **10**, puo' portare spesso a delle complicazioni.
Esistono svariate basi numeriche che soddisfano tutti i requisiti elencati in precedenza; le due basi che hanno avuto pero' maggiore diffusione sono la base **8** e la base **16**.

4.7.1 Sistema di numerazione posizionale in base b = 8

Il sistema di numerazione posizionale in base **8** viene chiamato **ottale**, e si ottiene a partire dall'insieme formato dai seguenti **8** simboli:
`S = {0, 1, 2, 3, 4, 5, 6, 7}`
In **Assembly** i numeri espressi in ottale sono contrassegnati dal suffisso **O** oppure **Q** (maiuscolo o minuscolo); e' preferibile usare la lettera **Q** perche' la lettera **O** puo' essere confusa con lo zero. Per convenzione, i numeri privi di suffisso si considerano espressi in base **10**; in caso di necessita' si puo' anche usare il suffisso **D** per i numeri in base **10**.
Come ormai gia' sappiamo, tutte le considerazioni svolte per le basi **2** e **10** si possono estendere con estrema facilita' alla base **8**; l'aspetto importante da ricordare e' che, dato un numero in base **8**, il peso di una cifra in posizione **p** e' espresso da $\mathbf{8^p}$.
Dimostriamo ora come il passaggio da base **8** a base **2** e viceversa sia immediato; a tale proposito, applichiamo le proprieta' matematiche dei sistemi di numerazione posizionali. Innanzi tutto osserviamo che per esprimere in base **2** tutti i numeri interi positivi da **0** a **7**, sono sufficienti **3** bit; infatti, con **3** bit possiamo rappresentare la sequenza:
`000b, 001b, 010b, 011b, 100b, 101b, 110b, 111b`
Questa sequenza corrisponde appunto ai numeri interi positivi da **0** a **7**; quindi, una qualsiasi cifra ottale puo' essere espressa attraverso **3** cifre binarie.
Osserviamo poi che in ottale la base **8** puo' essere espressa come $\mathbf{8=2^3}$; in base a queste osservazioni, possiamo effettuare in modo semplicissimo le conversioni da base **2** a base **8** e viceversa. La Figura 26 illustra il metodo di conversione da ottale a binario:

Figura 26 - Conversione da base 8 a base 2

```
Vogliamo convertire in binario il numero ottale 75q = 61d = 111101b

75q = 70q + 5q =
    = 7q * 10q + 5q * 1q =
    = 7 * 8^1 + 5 * 8^0 =
    = 111b * (2^3)^1 + 101b * (2^3)^0 =
    = (1 * 2^2 + 1 * 2^1 + 1 * 2^0) * 2^3 + (1 * 2^2 + 0 * 2^1 + 1 * 2^0) * 2^0 =
    = 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 =
    = 111101b
```

Osservando la Figura 26 possiamo constatare che per convertire rapidamente un numero ottale in binario, basta sostituire ad ogni cifra ottale la sua codifica in binario a **3** bit; tutto cio' e' una diretta conseguenza del fatto che **8=2^3**. Dato allora il numero ottale **75q**, possiamo osservare che la codifica binaria di **7q** e' **111b**, mentre la codifica binaria di **5q** e' **101b**; unendo i due codici binari si ottiene proprio **111101b**.
La conversione inversa (da binario a ottale) e' altrettanto immediata; basta applicare infatti il procedimento inverso a quello appena illustrato. Si prende quindi il numero binario, lo si suddivide in gruppi di tre cifre partendo da destra (aggiungendo se necessario zeri non significativi a sinistra), si sostituisce ogni gruppo di tre bit con la corrispondente cifra ottale e il gioco e' fatto.
Vediamo l'esempio della Figura 26 al contrario; prendiamo il numero binario **111101b** e suddividiamolo in gruppi di tre cifre ottenendo **111b 101b**. Osserviamo ora che **111b** in ottale corrisponde a **7q**, mentre **101b** in ottale corrisponde a **5q**; unendo le due cifre ottali si ottiene proprio **75q**.
Come si puo' notare, le conversioni da una base all'altra sono veramente rapide quando tra le due basi esiste la relazione matematica che abbiamo visto prima; per poter eseguire queste conversioni in modo veloce e sicuro, e' consigliabile munirsi di una apposita tabella di conversione tra le principali basi numeriche. Questa tabella, a dispetto della sua semplicita', offre un aiuto enorme ai programmatori.

4.7.2 Sistema di numerazione posizionale in base b = 16

Il sistema ottale garantisce una rappresentazione compatta dei numeri binari, e ci permette di passare facilmente da base **8** a base **2** o viceversa; purtroppo pero' i numeri ottali non garantiscono l'altro importante requisito, e cioe' la possibilita' per il programmatore di individuare facilmente dalla rappresentazione ottale il valore di ogni singolo bit. Proprio per questo motivo, la rappresentazione ottale e' stata quasi del tutto soppiantata dalla rappresentazione **esadecimale**; come si intuisce dal nome, si tratta di un sistema di numerazione posizionale in base **b=16** formato quindi dal seguente insieme di **16** simboli:
`S = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}`
Questi simboli equivalgono alle cifre decali comprese tra **0** e **15**.
Osserviamo innanzi tutto che in relazione alla base **16** si ha **16=2^4**; in base allora a quanto abbiamo visto per la base **8**, possiamo dire che per esprimere in binario una qualunque cifra esadecimale, dobbiamo utilizzare al massimo **4** bit. Con **4** bit possiamo esprimere tutti i numeri binari che vanno da **0000b** a **1111b**; questa sequenza corrisponde a tutti i numeri in base **10** compresi tra **0** e **15**. La stessa sequenza corrisponde quindi a tutti i numeri esadecimali compresi tra **0** e **F**; per convenzione, in **Assembly** tutti i numeri esadecimali recano il suffisso **H** (o **h**). La Figura 27 mostra un esempio di conversione da base **16** a base **2**:

Figura 27 - Conversione da base 16 a base 2

Vogliamo convertire in binario il numero esadecimale 9Bh = 155d = 10011011b

$9Bh = 90h + Bh =$
$= 9h * 10h + Bh * 1h =$
$= 9h * 16^1 + Bh * 16^0 =$
$= 1001b * (2^4)^1 + 1011b * (2^4)^0 =$
$= (1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0) * 2^4 + (1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0) * 2^0 =$
$= 1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 =$
$= 10011011b$

Osservando la Figura 27 possiamo constatare che per convertire rapidamente un numero esadecimale in binario, basta sostituire ad ogni cifra esadecimale la sua codifica in binario a **4** bit; tutto cio' e' una diretta conseguenza del fatto che **$16=2^4$**. Dato allora il numero esadecimale **9Bh**, possiamo osservare che la codifica binaria di **9h** e' **1001b**, mentre la codifica binaria di **Bh** e' **1011b**; unendo i due codici binari si ottiene proprio **10011011b**.
La conversione inversa (da binario a esadecimale) e' altrettanto immediata; basta applicare infatti il procedimento inverso a quello appena illustrato. Si prende quindi il numero binario, lo si suddivide in gruppi di quattro cifre partendo da destra (aggiungendo se necessario zeri non significativi a sinistra), si sostituisce ogni gruppo di quattro bit con la corrispondente cifra esadecimale e il gioco e' fatto.
Vediamo l'esempio della Figura 27 al contrario; prendiamo il numero binario **10011011b** e suddividiamolo in gruppi di quattro cifre ottenendo **1001b 1011b**. Osserviamo ora che **1001b** in esadecimale corrisponde a **9h**, mentre **1011b** in esadecimale corrisponde a **Bh**; unendo le due cifre esadecimali si ottiene proprio **9Bh**.
Ancora una volta si consiglia di avere sempre a portata di mano la tabella di conversione tra le principali basi numeriche che, tra l'altro, mostra in modo evidente la corrispondenza tra cifre o gruppi di cifre in basi diverse.
Il sistema di numerazione esadecimale permette di rappresentare i numeri binari in forma estremamente compatta, e garantisce anche una notevole semplicita' nelle conversioni da base **16** a base **2** e viceversa. Ma l'altro importante pregio del sistema di numerazione esadecimale, e' rappresentato dal fatto che un numero espresso in base **16** ci permette di intuire facilmente il valore di ogni suo singolo bit; considerando ad esempio il numero esadecimale **7ABFh**, si intuisce subito che i primi quattro bit (quelli piu' a destra) valgono **1111b**.
Il sistema di numerazione esadecimale rappresenta la vera soluzione al nostro problema, e per questo motivo viene utilizzato in modo massiccio non solo da chi programma in **Assembly**, ma anche da chi usa i linguaggi di programmazione di alto livello; tanto e' vero che tutti i linguaggi di programmazione supportano i numeri esadecimali. La Figura 28 illustra ad esempio la rappresentazione esadecimale del numero **7ABFh** secondo la sintassi dei principali linguaggi di alto livello:

Figura 28

Linguaggio	**Sintassi**
Assembly	7ABFh
C	0x7ABF
Pascal	$7ABF
BASIC	&H7ABF
FORTRAN	16#7ABF

Riassumendo le cose appena viste, possiamo dire che in generale, le conversioni numeriche da base **2** a base **b=2^k** e viceversa, sono immediate in quanto ogni cifra del numero in base **b** corrisponde a un gruppo di **k** cifre in binario.

4.7.3 Conversioni numeriche che coinvolgono la base 10.

Come abbiamo gia' visto, non e' possibile esprimere la base **10** sotto forma di potenza intera di **2**; spesso pero' capita di dover eseguire conversioni in base **10** da altre basi o viceversa, e quindi bisogna saper affrontare anche questo problema. La soluzione piu' semplice e sbrigativa consiste nel procurarsi una calcolatrice scientifica capace di maneggiare i numeri binari, ottali, esadecimali, etc, oltre che ovviamente i numeri decimali; queste calcolatrici mettono a disposizione funzioni di conversione istantanea da una base all'altra evitando la necessita' di eseguire calcoli spesso fastidiosi. E' anche possibile utilizzare la calcolatrice di **Windows** in modalita' scientifica.
Chi decide invece di affrontare direttamente queste conversioni, puo' rendersi conto che i calcoli da svolgere sono abbastanza semplici; la conversione di un numero da una base qualunque alla base **10** e' una ovvia conseguenza della struttura dei sistemi di numerazione posizionali. La Figura 29 mostra una serie di esempi pratici:

Figura 29 - Conversione da base b a base 10

Da base 2 a base 10:

01010101b = $0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$ =
= 0 * 128 + 1 * 64 + 0 * 32 + 1 * 16 + 0 * 8 + 1 * 4 + 0 * 2 + 1 * 1 =
= 0 + 64 + 0 + 16 + 0 + 4 + 0 + 1 = 85d

Da base 8 a base 10:

345q = $3 * 8^2 + 4 * 8^1 + 5 * 8^0$ =
= 3 * 64 + 4 * 8 + 5 * 1 = 192 + 32 + 5 = 229d

Da base 16 a base 10:

3C2Fh = $3 * 16^3 + C * 16^2 + 2 * 16^1 + F * 16^0$ =
= 3 * 4096 + 12 * 256 + 2 * 16 + 15 * 1 =
= 12288 + 3072 + 32 + 15 = 15407d

Per la conversione inversa, cioe' da base **10** a un'altra base **b**, si puo' utilizzare un algoritmo chiamato **metodo delle divisioni successive**. Si prende il numero da convertire e lo si divide ripetutamente per la base **b** (divisione intera) finche' il quoziente non diventa zero; i resti di tutte le divisioni eseguite rappresentano le cifre del numero nella nuova base **b** a partire dalla meno significativa. La Figura 30 mostra un esempio di conversione da base **10** a base **16**:

Figura 30 - Conversione da base 10 a base 16

```
Vogliamo convertire 15407 in base 16:

15407 / 16 = Q = 962, R = 15 = Fh
  962 / 16 = Q =  60, R =  2 = 2h
   60 / 16 = Q =   3, R = 12 = Ch
    3 / 16 = Q =   0, R =  3 = 3h
```

Disponendo ora in sequenza i resti di queste divisioni (a partire dall'ultimo) otteniamo **3C2Fh** che e' proprio la rappresentazione in base **16** di **15407**.

Raramente capita di dover passare da una base **b1** a una base **b2** entrambe diverse da **10** e tali che una non e' una potenza intera dell'altra; in questi casi, conviene usare la base **10** come base intermedia passando quindi da base **b1** a base **10** e da base **10** a base **b2** attraverso gli algoritmi appena illustrati. Per approfondire ulteriormente questi concetti, si consiglia di consultare un testo di elettronica digitale.

A questo punto, possiamo dire di aver raggiunto il grado di semplificazione necessario per implementare via hardware tutti i concetti esposti in questo capitolo; nel prossimo capitolo vedremo appunto quali tecniche vengono utilizzate per rappresentare fisicamente le informazioni all'interno del computer.

Capitolo 5 - La logica del computer

Nota importante:
Questa pagina contiene simboli appartenenti al set **UNICODE**; per la corretta visualizzazione di questi simboli in ambiente **Windows**, e' necessario disporre del font **Lucida Sans Unicode**, o in alternativa **Arial Unicode MS**.
Nei precedenti capitoli abbiamo raggiunto una notevole semplificazione nella codifica delle informazioni da rappresentare sul computer; in particolare, abbiamo visto che utilizzando il sistema di numerazione binario, possiamo codificare qualsiasi informazione attraverso i due soli simboli **0** e **1**. La conseguenza piu' importante di tutto cio' e' che i circuiti del computer dovranno essere in grado di gestire soltanto questi due simboli anziche' ad esempio i dieci simboli del sistema di numerazione in base **10**; lo scopo di questo capitolo e' quello di illustrare le tecniche basilari che permettono ai circuiti elettronici di un computer di gestire fisicamente il codice binario.

5.1 Segnali analogici e segnali logici

E' ovvio che un computer, essendo un oggetto inanimato e privo di intelligenza propria, non e' in grado di capire veramente cosa sia un numero, e questo puo' sembrare un ostacolo piuttosto difficile da superare; grazie pero' al fatto che una cifra binaria puo' assumere solo i due valori **0** e **1**, possiamo risolvere questo problema associando questi due simboli (bit) a due situazioni fisiche distinte, come ad esempio i due stati fisici possibili per un interruttore elettrico (on/off), due valori di temperatura, due valori di una intensita' di corrente elettrica, due valori di una tensione elettrica, etc.
Visto che abbiamo a che fare con un circuito elettronico, appare ovvio orientarsi sulle grandezze elettriche come tensioni e correnti, e quindi in generale, su segnali elettrici che attraversano i circuiti del computer; possiamo ad esempio scegliere due valori distinti di tensione assunti da un segnale elettrico, associando a questi due valori i due simboli **0** e **1**. Escludendo i segnali costanti nel tempo che quindi possono assumere un solo valore, dobbiamo fare riferimento necessariamente ai segnali che variano nel tempo assumendo due o piu' valori distinti. I segnali che attraversano un circuito elettrico possono essere suddivisi in due grandi categorie: **segnali analogici** e **segnali logici**; la Figura 1 illustra questi due tipi di segnale:

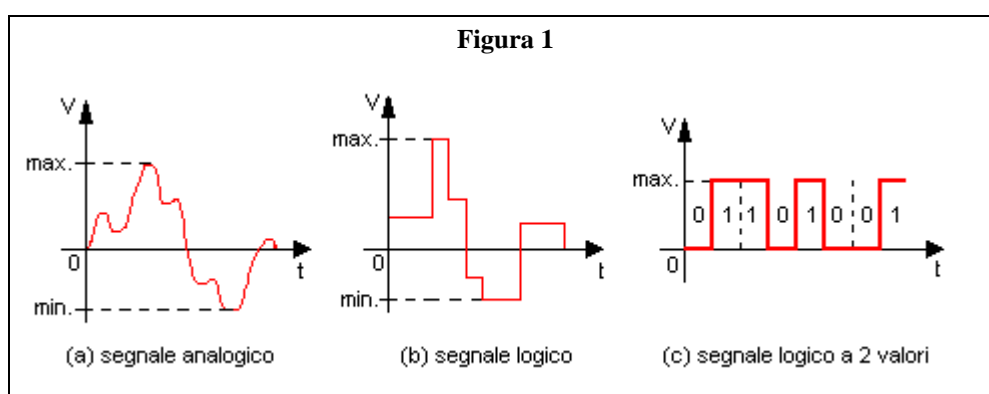



Figura 1

Si definisce **analogico** un segnale che varia nel tempo assumendo con continuita' tutti gli infiniti valori compresi tra un massimo ed un minimo; in Figura 1a possiamo vedere ad esempio un segnale elettrico analogico che al trascorrere del tempo **t** assume in modo continuo (cioe' senza salti) tutti gli infiniti valori di tensione elettrica (**V**) compresi tra **max** e **min**.
Si definisce **logico** un segnale che varia nel tempo assumendo un numero finito di valori compresi tra un massimo ed un minimo; la Figura 1b e la Figura 1c mostrano due esempi di segnali elettrici logici che al trascorrere del tempo **t** assumono solo alcuni dei valori di tensione elettrica (**V**)

compresi tra **max** e **min**. Nel caso particolare della Figura 1c, il segnale logico assume due soli valori, e cioe' il valore minimo **0** e il valore massimo **max**; in generale, per i segnali logici possiamo assumere che teoricamente il passaggio da un valore all'altro avvenga in modo istantaneo, e cioe' in un tempo nullo.
A questo punto appare evidente il fatto che i segnali logici sono quelli piu' indicati per risolvere il nostro problema; osserviamo infatti che e' possibile associare un certo numero di simboli da codificare, ad altrettanti livelli di tensione (detti **livelli logici**) assunti da questa categoria di segnali elettrici. Consideriamo in particolare proprio il segnale logico di Figura 1c che puo' assumere nel tempo solo due valori distinti di tensione (che necessariamente coincideranno con il minimo e il massimo); come e' stato gia' detto, assumiamo per semplicita' che il passaggio da un livello all'altro avvenga in modo istantaneo (tempo di commutazione nullo). Possiamo far coincidere ad esempio il livello piu' basso con il potenziale elettrico della massa del circuito; indicando con **GND** il potenziale di massa, possiamo porre ad esempio:

```
GND = 0 volt
```

Analogamente, possiamo far coincidere il livello piu' alto con il potenziale elettrico positivo di alimentazione del circuito; indicando con **+Vcc** il potenziale di alimentazione, possiamo porre ad esempio:

```
+Vcc = +5 volt
```

In elettronica digitale, i due livelli logici minimo e massimo assunti dal segnale di Figura 1c, vengono indicati con diversi nomi come ad esempio **basso/alto**, **low/high**, **off/on**, **GND/+Vcc** e cosi' via. Naturalmente, in relazione al sistema di numerazione binario, e' intuitivo il fatto che le cifre binarie **0** e **1** verranno associate rispettivamente a **GND** e **+Vcc**; come vedremo nel seguito del capitolo e nei capitoli successivi, con questo accorgimento il computer si trova ad elaborare segnali elettrici senza rendersi conto che questi segnali rappresentano (per noi esseri umani) dei numeri.
I circuiti elettronici che elaborano i segnali logici, prendono il nome di **circuiti logici** o **reti logiche**; un problema molto delicato che e' necessario affrontare nei circuiti logici, consiste nel tenere adeguatamente "distanti" il livello logico **0** dal livello logico **1** in modo da evitare commutazioni indesiderate dovute ad esempio a disturbi elettrici esterni. Si tratta quindi di scegliere due valori di tensione che risultino non troppo vicini tra loro; contemporaneamente pero' si deve anche rispettare il vincolo rappresentato dalla tensione di alimentazione **+Vcc** del circuito logico. Avendo a che fare con due soli simboli da rappresentare, possiamo associare un simbolo a **GND** e l'altro simbolo a **+Vcc** ottenendo cosi' la massima distanza possibile tra i due livelli logici; in moltissimi circuiti logici si utilizza un valore di **+Vcc** pari a **+5V**.
Le considerazioni appena esposte ci fanno capire che non avrebbe molto senso realizzare un computer capace di lavorare in base **10**; in un caso del genere infatti andremmo incontro ad enormi complicazioni circuitali, e si presenterebbe anche il problema della eccessiva vicinanza tra due livelli logici adiacenti. Questa situazione farebbe aumentare notevolmente il rischio di commutazioni indesiderate; la Figura 2 ad esempio illustra quello che succederebbe se si dovessero gestire **10** livelli logici differenti con una **+Vcc** di **+5V**:

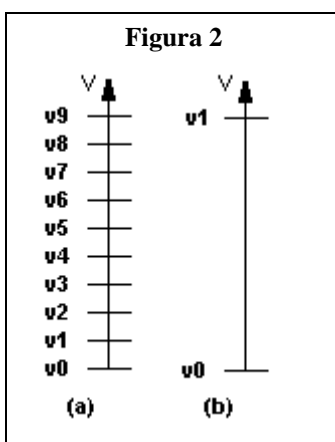
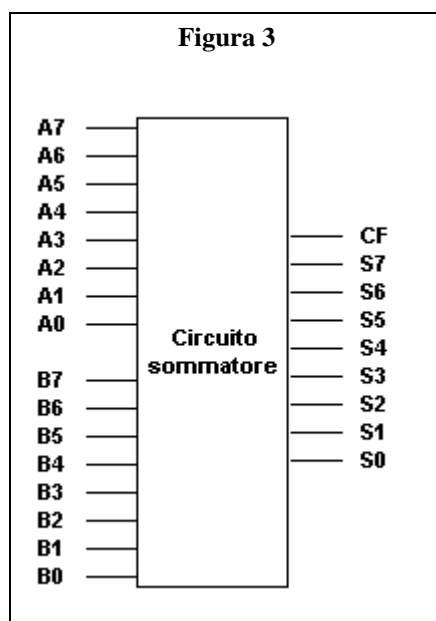



Figura 2

In Figura 2a vediamo i **10** livelli di tensione necessari per rappresentare i **10** diversi simboli del sistema di numerazione posizionale in base **10**; in Figura 2b vediamo invece i **2** livelli di tensione necessari per rappresentare i **2** diversi simboli del sistema di numerazione posizionale in base **2**.
Nel caso del codice binario abbiamo **v0=0V** e **v1=+5V**; in questo modo tra i due unici livelli logici esiste una distanza di **+5V** che e' anche la massima distanza possibile. Nel caso invece del codice decale, per gestire tutti i **10** differenti simboli dobbiamo necessariamente suddividere l'intervallo tra **0V** e **+5V** in **9** parti uguali, ciascuna delle quali rappresenta un salto di appena **0.55V**; appare evidente quindi che una distanza cosi' piccola tra due livelli logici adiacenti comporterebbe un altissimo rischio di commutazioni indesiderate.

5.2 I dispositivi logici

I segnali logici che transitano nei circuiti del computer, vengono gestiti ed elaborati attraverso appositi dispositivi chiamati **dispositivi logici**; si usa anche la definizione di **dispositivi digitali**. Il termine "logico" e' legato al fatto che questi dispositivi producono un segnale in uscita che e' una "logica" conseguenza del segnale (o dei segnali) in ingresso; il termine "digitale" e' legato invece al fatto che i segnali da elaborare rappresentano come sappiamo le cifre (digits) di un numero binario.
Consideriamo ora un codice binario a **8** bit come ad esempio **01010110b**; per gestire gli **8** bit di questo numero il computer si serve di **8** apposite linee elettriche che nel loro insieme formano un cosiddetto **BUS**. Su ciascuna linea transita uno dei bit del codice binario; partendo allora dal **LSB** possiamo dire che sulla prima linea transita un livello logico **0**, sulla seconda linea transita un livello logico **1**, sulla terza linea transita un livello logico **1** e cosi' via.
Se vogliamo realizzare allora un dispositivo logico capace di sommare tra loro due numeri binari da **8** bit ciascuno, dobbiamo fare in modo che il dispositivo stesso sia dotato delle necessarie linee di ingresso e di uscita; la Figura 3 illustra in modo schematico la struttura di un dispositivo di questo genere:

Figura 3

Come si puo' notare, abbiamo **16** linee di ingresso attraverso le quali entrano i due numeri binari da sommare; lungo le linee da **A0** a **A7** entrano gli **8** bit del primo addendo, mentre lungo le linee da **B0** a **B7** entrano gli **8** bit del secondo addendo. Abbiamo poi **8** linee di uscita (da **S0** a **S7**) lungo le quali esce il numero binario che rappresenta la somma dei due numeri in ingresso; la nona linea (**CF**) contiene il riporto della somma.
La Figura 3 ci permette anche di capire meglio il concetto di architettura di una **CPU** (in questo

caso si tratta chiaramente di architettura a **8** bit); appare evidente il fatto che con il dispositivo di Figura 3 non e' possibile sommare via hardware numeri binari formati da piu' di **8** bit.
Si puo' anche constatare che un dispositivo sommatore appartenente ad una **CPU** con architettura a **16** bit sara' dotato di **32** linee di ingresso e **16+1** linee di uscita; al crescere del numero di linee disponibili cresce la potenza di calcolo della **CPU**, ma parallelamente cresce anche la complessita' circuitale.

In definitiva possiamo dire che un dispositivo logico riceve in ingresso (**input**) una serie di segnali digitali, e produce in uscita (**output**) un'altra serie di segnali digitali che sono una logica conseguenza dei segnali in ingresso; a questo punto resta da capire come sia possibile eseguire su questi segnali digitali delle operazioni matematiche che producono dei risultati sempre sotto forma di segnali digitali. Le tecniche che vengono impiegate nella realizzazione dei circuiti logici del computer si basano principalmente sull'algebra sviluppata nel **XVIII** secolo dal matematico **George Boole** che non poteva certo immaginare che un giorno i suoi studi avrebbero avuto una applicazione pratica cosi' importante.

5.3 L'algebra di Boole

L'algebra di **Boole**, definisce le operazioni fondamentali che si possono compiere sugli insiemi formati da un unico elemento che puo' essere **0** o **1**; l'algebra di **Boole** e' fortemente basata sulla **teoria degli insiemi**, per cui e' necessario definire le nozioni fondamentali relative agli insiemi stessi.
Il concetto di **insieme** viene considerato primitivo in quanto e' naturale associare questo termine ad un gruppo o aggregato di oggetti della stessa natura come ad esempio: un gruppo di automobili, un gruppo di sedie, un gruppo di bicchieri, un gruppo di dischi, un gruppo di computers, etc; analogamente, anche il concetto di **elemento di un insieme** e' primitivo in quanto rappresenta ovviamente un oggetto appartenente all'insieme stesso.
Gli insiemi vengono rappresentati con le lettere maiuscole (**A, B, C, ...**) , mentre gli elementi di un insieme vengono rappresentati con le lettere minuscole (**a, b, c, ...**); un insieme **A** formato dai quattro elementi **a, b, c, d** viene rappresentato come:
$A = \{a, b, c, d\}$
Un elemento puo' appartenere o meno ad un insieme; se l'elemento **a** appartiene all'insieme **A** si scrive:
$a \in A$
Se invece **a** non appartiene ad **A** si scrive:
$a \notin A$
Dato un insieme **A**, si dice che l'insieme **B** e' un **sottoinsieme** di **A** se tutti gli elementi di **B** appartengono ad **A** e si scrive:
$B \subseteq A$, oppure $A \supseteq B$

Se $B \subseteq A$ ed esiste almeno un elemento di **A** che non appartiene a **B** si dice che **B** e' un **sottoinsieme proprio** di **A** e si scrive:
$B \subset A$, oppure $A \supset B$

Se $B \subseteq A$ e $A \subseteq B$, allora tutti gli elementi di **A** appartengono a **B** e viceversa e quindi i due insiemi sono uguali e si scrive:
$A = B$
Un insieme privo di elementi si chiama **insieme vuoto** e viene rappresentato dal simbolo $\varnothing$.
Se consideriamo ad esempio l'insieme delle automobili targate **Roma**, abbiamo stabilito una proprieta' chiamata **proprieta' caratteristica** che ci dice in modo inequivocabile se un'automobile appartiene o meno al nostro insieme (solo se la macchina e' targata **Roma** appartiene al nostro

insieme); questo insieme e' "immerso" nell'insieme piu' vasto che comprende tutte le automobili del mondo chiamato **insieme universale**. Questo concetto e' valido in generale per qualsiasi insieme composto da qualsiasi tipo di elementi; l'insieme universale viene indicato con U.
Un insieme composto da un numero finito di elementi (o da una infinita' numerabile di elementi) si chiama **insieme discreto**; se invece l'insieme e' costituito da una infinita' non numerabile di elementi, allora si parla di **insieme continuo**.

Vediamo ora le operazioni che si possono compiere sugli insiemi.

Dati due insiemi **A** e **B**, si definisce **unione** o **somma** dei due insiemi, un nuovo insieme costituito dagli elementi che appartengono ad **A** o a **B**; questa operazione che viene rappresentata come:
$A \cup B$
si estende in generale ad un numero qualsiasi di insiemi.
Ad esempio, se:
$A = \{5, 7, 9\}$ e $B = \{7, 9, 15, 20, 24\}$, allora $A \cup B = S = \{5, 7, 9, 15, 20, 24\}$

Dati due insiemi **A** e **B**, si definisce **intersezione** o **prodotto** dei due insiemi, un nuovo insieme costituito dagli elementi che appartengono contemporaneamente sia ad **A** che a **B**; questa operazione che viene rappresentata come:
$A \cap B$
si estende in generale ad un numero qualsiasi di insiemi.
Ad esempio, se:
$A = \{5, 7, 9\}$ e $B = \{7, 9, 15, 20, 24\}$, allora $A \cap B = P = \{7, 9\}$

Se invece:
$A = \{1, 2, 3\}$ e $B = \{4, 5, 6\}$, allora $A \cap B = P = \varnothing$
e i due insiemi si dicono **disgiunti**.

Dato un insieme **A**, si definisce **complemento** di **A** rispetto all'insieme universale U, l'insieme formato da tutti gli elementi di U che non appartengono da **A**; questa operazione viene rappresentata con $\bar{A}$.
Ad esempio, se **A** e' l'insieme delle automobili targate **Roma**, allora il complemento di **A** e' l'insieme di tutte le automobili del mondo che non sono targate **Roma**.
In base alle considerazioni appena esposte, sono facilmente verificabili le uguaglianze mostrate in Figura 4:

Figura 4

Unione	Intersezione
$A \cup A = A$	$A \cap A = A$
$A \cup \varnothing = A$	$A \cap \varnothing = \varnothing$
$A \cup U = U$	$A \cap U = A$
$A \cup \bar{A} = U$	$A \cap \bar{A} = \varnothing$
$\varnothing \cup U = U$	$\varnothing \cap U = \varnothing$

5.4 Concetti fondamentali dell'algebra di Boole

Vediamo ora i concetti e i postulati fondamentali dell'algebra di **Boole** che in seguito verranno interpretati e dimostrati; ricordiamo che l'algebra di **Boole** si applica agli insieme formati da un unico elemento che puo' essere **0** oppure **1**. I concetti fondamentali dell'algebra di **Boole** sono elencati in Figura 5:

Figura 5

Nome	Simbolo	Significato
CLASSE	K	e' l'insieme formato dai due unici elementi 0, 1
OR	+	(somma binaria) e' una operazione definita su due elementi in classe K
AND	·	(prodotto binario) e' una operazione definita su due elementi in classe K
NOT	¯	(negazione unaria) e' una operazione definita su un solo elemento in classe K
=	=	(uguale) definisce l'uguaglianza tra elementi in classe K

Per chiarire meglio questi concetti, consideriamo un insieme **A** formato da un unico elemento che puo' essere **0** oppure **1**; appare evidente che un insieme del genere e' un sottoinsieme proprio dell'insieme **K**. Si dice allora che **A** e' un **elemento in classe K**.
Gli operatori come **OR** (somma binaria) e **AND** (prodotto binario) si chiamano **operatori binari** in quanto richiedono come operandi due elementi in classe **K**; l'operatore **NOT** (negazione unaria) si chiama invece **operatore unario** in quanto richiede un solo operando che ovviamente deve essere un elemento in classe **K**.

5.5 Interpretazione dei concetti fondamentali

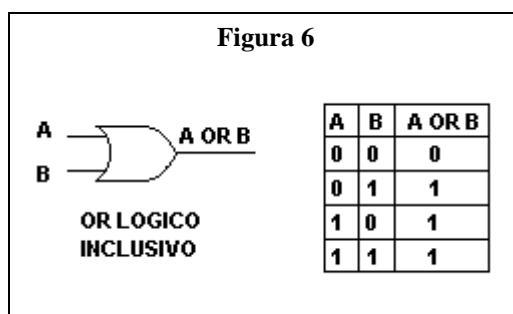
Vediamo ora di interpretare i concetti fondamentali dell'algebra di **Boole** servendoci della cosiddetta **algebra delle proposizioni**; ci serviamo cioe' di dichiarazioni che possono risultare **vere** o **false**. Questo significa che le dichiarazioni devono essere prive di qualsiasi ambiguita' e per questo motivo si parla anche di **proposizioni logiche**.
Prendiamo ad esempio le due dichiarazioni:
`"La capitale dell'Italia e' Roma"` e `"La capitale della Francia e' Berlino"`
La prima dichiarazione e' vera (**TRUE**) mentre la seconda e' chiaramente falsa (**FALSE**) e questo puo' essere detto senza ombra di dubbio.
Invece la dichiarazione:
`"L'Italia e' piu' bella della Francia"`
non e' una proposizione logica in quanto la risposta e' influenzata dai gusti personali.
Tornando alle proposizioni logiche, associamo alla proposizione vera il simbolo **1** (**TRUE**) e alla proposizione falsa il simbolo **0** (**FALSE**); stabilite queste convenzioni, possiamo studiare i concetti **OR**, **AND**, **NOT** e =, costruendo per ciascuno di essi, la cosiddetta **tabella della verita'** o **truth table** che ci da' una rappresentazione visiva dei risultati che si ottengono applicando questi operatori agli elementi **0** e **1**.

5.5.1 OR logico

Date le due proposizioni **A** e **B**, la composizione delle due proposizioni attraverso l'**OR logico** si scrive:
`A OR B`
e `risulta vera se almeno una delle due proposizioni e' vera`; componendo ad esempio le due dichiarazioni viste prima:
`"La capitale dell'Italia e' Roma" OR "La capitale della Francia e' Berlino"`
si ottiene **1** (**TRUE**) in quanto la prima delle due dichiarazioni e' vera.

Nella Figura 6, vediamo il simbolo logico dell'**OR** e la relativa tabella della verita':

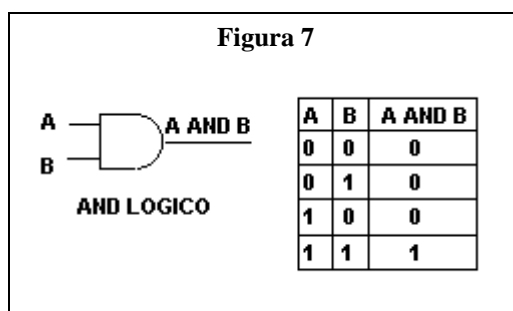
Figura 6

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

La tabella della verita' in genere viene letta come: **0 OR 0 = 0, 0 OR 1 = 1** e cosi' via. A sinistra della tabella della verita' in Figura 6 vediamo il simbolo che schematizza il componente che nei circuiti elettronici del computer svolge il ruolo di **OR logico**; come si puo' notare, alla sinistra del componente sono presenti due linee di ingresso, mentre alla sua destra c'e' una linea di uscita e questo perche', essendo l'**OR** un operatore binario, ricevera' in input due segnali producendo un segnale in output.
In base alla tabella della verita', possiamo vedere che se in input entrano due segnali a livello logico **0**, il segnale in uscita e' anch'esso a livello logico **0**; se un segnale in ingresso e' a livello logico **0** mentre l'altro e' a **1**, il segnale in uscita sara' a livello logico **1** e cosi' via.
Facendo riferimento alla teoria degli insiemi, possiamo dire che l'**OR logico** corrisponde alla operazione di **unione** tra i due insiemi **A** e **B**; questo componente elettronico, fa' parte, insieme a quelli che seguono e a molti altri, della grande famiglia delle cosiddette **porte logiche**.
Le considerazioni appena esposte possono essere estese nel caso piu' generale possibile a una porta logica **OR** formata da un numero arbitrario **n** di ingressi e da una sola uscita. In questo caso, l'uscita fornisce un livello logico **1** quando almeno uno degli **n** ingressi e' a livello logico **1**; se tutti gli ingressi sono a livello logico **0**, anche in uscita si ottiene **0**.

5.5.2 AND logico

Date le due proposizioni **A** e **B**, la composizione delle due proposizioni attraverso l'**AND logico** si scrive:
`A AND B`
e `risulta vera solo se entrambe le proposizioni sono vere`; componendo ad esempio le due dichiarazioni viste prima:
`"La capitale dell'Italia e' Roma" AND "La capitale della Francia e' Berlino"`
si ottiene **0** (**FALSE**) in quanto la seconda delle due dichiarazioni e' falsa.
Nella Figura 7, vediamo il simbolo logico dell'**AND** e la relativa tabella della verita':

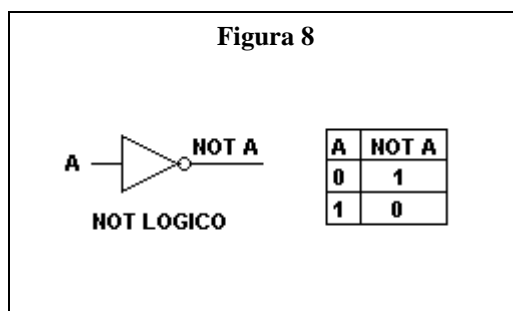
Figura 7

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Attraverso la Figura 7 si verifica subito che solo quando i due segnali in ingresso sono entrambi a livello logico **1** (cioe' quando le due proposizioni sono entrambe vere), l'**AND logico** produrra' in uscita un segnale a livello logico **1**; in tutti gli altri casi il segnale in uscita sara' invece a livello

logico **0**.
La figura 7 mostra chiaramente il significato di **operatore binario**; anche l'**AND** infatti, (come l'**OR**) opera su due elementi in classe **K** per produrre il risultato finale.
Si puo' facilmente dimostrare che nella teoria degli insiemi, l'**AND** coincide con l'operazione di **intersezione** tra i due insiemi **A** e **B**.
Le considerazioni appena esposte possono essere estese nel caso piu' generale possibile a una porta logica **AND** formata da un numero arbitrario **n** di ingressi e da una sola uscita. In questo caso, l'uscita fornisce un livello logico **1** quando tutti gli **n** ingressi sono a livello logico **1**; in tutti gli altri casi (cioe' quando almeno uno degli **n** ingressi e' a **0**), in uscita si ottiene **0**.

5.5.3 NOT logico

A differenza degli operatori binari, il **NOT logico** e' un operatore **unario** perche' opera su di un solo elemento in classe **K**. Data la proposizione **A**, la negazione di questa proposizione si scrive:
`NOT A`
e `risulta vera se` **`A`** `e' falsa`, mentre `risulta falsa se` **`A`** `e' vera`.
Nel caso ad esempio di:
`NOT "La capitale dell'Italia e' Roma"`
si ottiene **0** (**FALSE**) in quanto stiamo negando una proposizione che e' vera; questa negazione produce infatti la proposizione:
`"La capitale dell'Italia non e' Roma"`
che e' chiaramente falsa.
Nel caso invece di:
`NOT "La capitale della Francia e' Berlino"`
si ottiene **1** (**TRUE**) in quanto stiamo negando una proposizione che e' falsa; questa negazione produce infatti la proposizione:
`"La capitale della Francia non e' Berlino"`
che e' chiaramente vera.
Nella Figura 8, vediamo il simbolo logico del **NOT** e la relativa tabella della verita':

A	NOT A
0	1
1	0

La Figura 8 ci permette di constatare che il **NOT logico** ha una sola linea in ingresso e una sola linea in uscita. Se in ingresso entra un segnale a livello logico **0**, avremo in uscita un segnale a livello logico **1**; viceversa, se in ingresso entra un segnale a livello logico **1**, avremo in uscita un segnale a livello logico **0**.
Negli schemi dei circuiti elettronici spesso, il **NOT logico** viene rappresentato attraverso il solo cerchietto che si vede in Figura 8 sul vertice destro del triangolo.
Nella teoria degli insiemi, il **NOT logico** corrisponde all'operazione di **complemento** di **A** rispetto all'insieme universale **U** (in questo caso **U = {0, 1}**); si ottengono quindi tutti gli elementi di **U** esclusi quelli che appartengono ad **A**.

5.5.4 Uguaglianza logica (=)

L'uguaglianza logica, non e' associata a nessun componente elettronico in quanto esprime solamente il fatto che due elementi in classe **K** sono uguali tra loro; possiamo dire allora che due proposizioni logiche **A** e **B** sono uguali quando hanno la stessa tabella della verita'.
Nella teoria degli insiemi tutto questo discorso e' rappresentato dal fatto che:
B ⊆ A e A ⊆ B
Quindi, tutti gli elementi di **A** sono anche elementi di **B** e viceversa.

Una volta interpretati i concetti fondamentali, possiamo passare alla definizione e dimostrazione dei postulati dell'algebra di **Boole**.

5.6 Postulati fondamentali dell'algebra di Boole

1) Dati i due elementi **0** e **1** in classe **K**, allora le operazioni **OR** e **AND** applicate a questi elementi, saranno a loro volta in classe **K**, cioe' produrranno un risultato binario.
2) Se **A** e **B** sono due elementi in classe **K**, valgono le due uguaglianze:
A OR B = B OR A e A AND B = B AND A
In sostanza, anche per l'algebra di **Boole** e' valida la **proprieta' commutativa**.
3) Dati gli elementi **A**, **B** e **C** in classe **K**, valgono le uguaglianze:
A OR (B AND C) = (A OR B) AND (A OR C) e A AND (B OR C) = (A AND B) OR (A AND C)
Quindi, nell'algebra di **Boole** la **proprieta' distributiva** vale non solo per il prodotto rispetto alla somma, ma anche per la somma rispetto al prodotto.
4) L'**elemento neutro** per l'**OR** e' **0** e quindi:
0 OR A = A
qualunque sia il valore di **A**.
L'elemento neutro per l'**AND** e' **1** e quindi:
1 AND A = A
indipendentemente dal valore di **A**.
5) Per ogni elemento **A** in classe **K** valgono le due uguaglianze:
A OR Ā = 1 e A AND Ā = 0.

5.7 Dimostrazione dei postulati fondamentali

Grazie a quanto abbiamo appreso riguardo all'interpretazione dei **concetti fondamentali**, la dimostrazione dei **postulati** risulta piuttosto semplice.
Partiamo dal postulato n. **1** che afferma in pratica che applicando gli operatori **OR** e **AND** agli elementi **0** e **1**, si ottiene un risultato che puo' essere o **0** o **1**; per dimostrare questo postulato basta osservare le tabelle della verita' dei due operatori **OR** e **AND** (Figura 6 e Figura 7) e si vede subito che il risultato che si ottiene e' sempre, o **0** o **1**.
Il postulato n. **2** afferma che i due operatori **OR** e **AND** godono della **proprieta' commutativa**, e questo significa che si possono scambiare tra loro i due operandi (**A** e **B**) ottenendo gli stessi risultati; per dimostrarlo basta prendere le **tabelle della verita'** di Figura 6 e Figura 7, e scambiare tra loro i due elementi **A** e **B** ricalcolando poi **B OR A** e **B AND A** e verificando in tal modo che i risultati ottenuti sono identici a quelli di **A OR B** e **A AND B**.
Il postulato n. **3** afferma che nell'algebra di **Boole** vale la **proprieta' distributiva**, sia del prodotto rispetto alla somma, sia della somma rispetto al prodotto, mentre come sappiamo, nell'algebra tradizionale vale solo la proprieta' distributiva del prodotto rispetto alla somma; ad esempio:
3 x (2 + 5) = 21 = (3 x 2) + (3 x 5)
Per dimostrare il postulato n. **3** utilizziamo le tabelle della verita' di Figura 9 e di Figura 10:

Figura 9 - A OR (B AND C) = (A OR B) AND (A OR C)

A	B	C	B AND C	A OR (B AND C)	A OR B	A OR C	(A OR B) AND (A OR C)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Figura 10 - A AND (B OR C) = (A AND B) OR (A AND C)

A	B	C	B OR C	A AND (B OR C)	A AND B	A AND C	(A AND B) OR (A AND C)
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

In entrambe le tabelle si vede che la colonna 5 coincide con la colonna 8.
Il postulato n. **4** afferma che esiste l'**elemento neutro** sia per l'**OR** che per l'**AND**. Nell'algebra tradizionale, l'elemento neutro per la somma e' **0** in quanto, sommando un qualsiasi numero **n** a **0**, si ottiene sempre **n**; infatti:

`n + 0 = n`

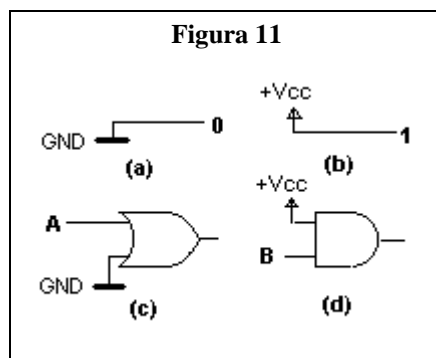
Per il prodotto invece, l'elemento neutro e' **1** in quanto, moltiplicando un qualsiasi numero **n** per **1**, si ottiene sempre **n**; infatti:

`n x 1 = n`

Per dimostrare questo postulato, basta servirsi ancora delle tabelle della verita' riferite ad un elemento **A** che puo' valere **0** o **1**; si verifica subito che:

`A OR 0 = A` e `A AND 1 = A`

E' importante sottolineare che **0** equivale ad una proposizione logica sempre falsa, mentre **1** equivale ad una proposizione logica sempre vera; nei circuiti digitali, si ottiene la proposizione logica sempre falsa, collegando un ingresso della porta logica a massa (**GND**) che corrisponde al livello logico basso (Figura 11a), mentre per avere una proposizione logica sempre vera si collega un ingresso della porta logica ad un punto del circuito a potenziale elettrico **+Vcc** che corrisponde al livello logico alto (Figura 11b).

Figura 11

Nei circuiti digitali inoltre, puo' capitare che uno o piu' ingressi di una porta logica non vengano utilizzati; questi ingressi devono essere ugualmente collegati al circuito per garantire il corretto funzionamento della porta logica stessa. Tenendo conto della funzione svolta dalle porte **OR** e **AND**, e' facile capire che: `ogni ingresso non utilizzato di una porta` **`OR`** `va' collegato a` **`GND`** (Figura 11c), mentre `ogni ingresso non utilizzato di una porta` **`AND`** `va' collegato a` **`+Vcc`** (Figura 11d).
E' evidente infatti che in caso contrario, si possono ottenere sull'uscita di queste porte, risultati imprevisti, come si puo' facilmente verificare.

Il postulato n. **5** infine e' abbastanza intuitivo, infatti se **A** vale **0**, allora **NOT A** vale **1** e viceversa; questo significa che abbiamo in ogni caso, una proposizione logica vera e una falsa. La conseguenza e' che:
`A OR (NOT A) = 1`, mentre `A AND (NOT A) = 0`

5.8 Teoremi dell'algebra di Boole

Un insieme di porte logiche collegate tra loro, formano una cosiddetta **rete combinatoria** o **R.C.**; quando il numero di porte logiche comincia a diventare piuttosto consistente, le **R.C.** tendono ad assumere una notevole complessita', e in questo caso, e' necessario trovare il modo di semplificare la rete stessa. I teoremi dell'algebra di **Boole** hanno proprio lo scopo di studiare, risolvere e semplificare le **R.C.**; per maggiori dettagli su questo argomento si puo' consultare un testo di elettronica digitale.
A titolo di esempio, citiamo solo il **teorema dell'idempotenza** che afferma che:
`A OR A = A` e `A AND A = A`
La verifica attraverso le **tabelle della verita'** e' immediata.
In generale l'elemento **A** puo' essere anche il risultato di una qualunque espressione booleana come ad esempio:
`A = B OR (C AND D)`
Attraverso questo teorema e' possibile ottenere notevoli semplificazioni delle **R.C**. Supponiamo ad esempio di voler realizzare attraverso porte logiche la funzione:
`f = (A AND B AND C) OR (B AND (NOT C)) OR (A AND B AND C) OR ((NOT A) AND B) OR (B AND (NOT C))`
Questa funzione richiede **1 OR** a **5** ingressi, **1 AND** a **3** ingressi, **3 AND** a **2** ingressi e **2 NOT**, per un totale di **8** porte logiche.
Raggruppiamo i termini simili (proprieta' associativa) e riscriviamo la funzione come:
`f = ((A AND B AND C) OR (A AND B AND C)) OR ((B AND (NOT C)) OR (B AND (NOT C))) OR ((NOT A) AND B)`
Applicando il teorema dell'idempotenza, notiamo subito che:
`(A AND B AND C) OR (A AND B AND C) = (A AND B AND C)`
e:
`(B AND (NOT C)) OR (B AND (NOT C)) = (B AND (NOT C))`
Possiamo allora riscrivere la funzione come:

```
f = (A AND B AND C) OR (B AND (NOT C)) OR ((NOT A) AND B)
```

La nuova funzione cosi' ottenuta richiede **1 OR** a **3** ingressi, **1 AND** a **3** ingressi, **2 AND** a **2** ingressi e **2 NOT**, per un totale di **6** porte logiche. La Figura 12 mostra le **R.C.** relative alle due funzioni:

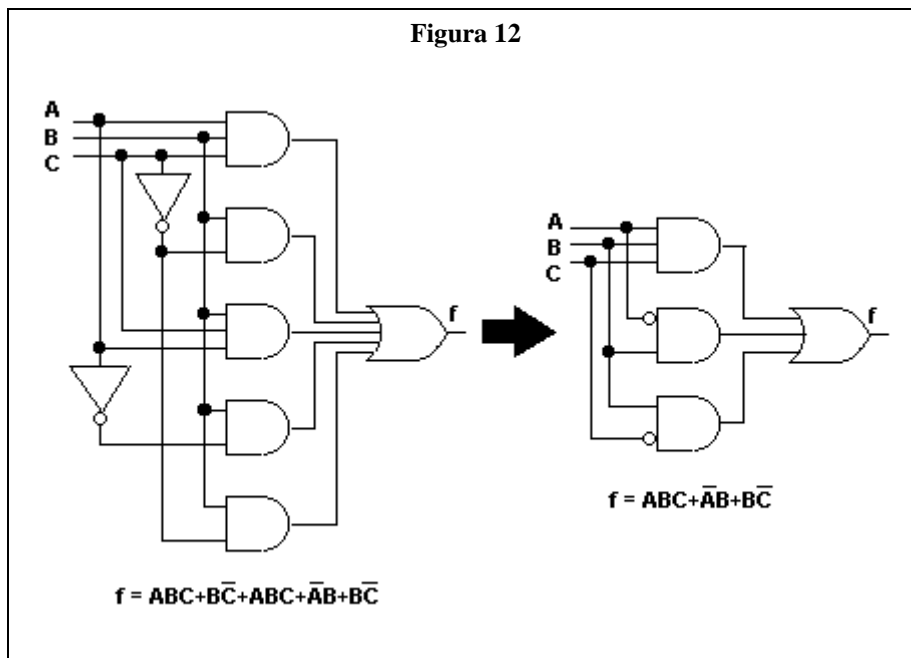



Figura 12

In Figura 12 a sinistra vediamo la **R.C.** della funzione non semplificata, mentre sulla destra vediamo la **R.C.** equivalente che si ottiene dai calcoli illustrati in precedenza; con l'ausilio di questi due circuiti si verifica subito che assegnando qualunque terna di valori agli elementi in ingresso **A, B, C**, si ottiene lo stesso valore in uscita per entrambe le **R.C.**. Si noti che nel circuito a destra la porta **NOT** e' stata sostituita (come si fa' usualmente) con un cerchietto; inoltre, il segno **+** indica l'operatore **OR**, mentre l'operatore **AND** (·) e' stato omesso come si fa' usualmente con il segno di moltiplicazione.

5.9 Altre porte logiche importanti

Le porte logiche **OR**, **AND**, **NOT**, vengono definite **fondamentali** in quanto tutte le altre porte logiche utilizzate nei circuiti digitali, possono essere ottenute a partire da queste tre; analizziamo le principali porte logiche derivate dalle tre fondamentali.

5.9.1 Porta OR Esclusivo

In figura 13, vediamo la porta **EX-OR** chiamata anche **Exclusive OR** (**OR Esclusivo**); si tratta di un operatore binario che restituisce un livello logico **1** solo quando i due segnali in ingresso hanno livelli logici diversi tra loro:

A	B	A EX-OR B
0	0	0
0	1	1
1	0	1
1	1	0

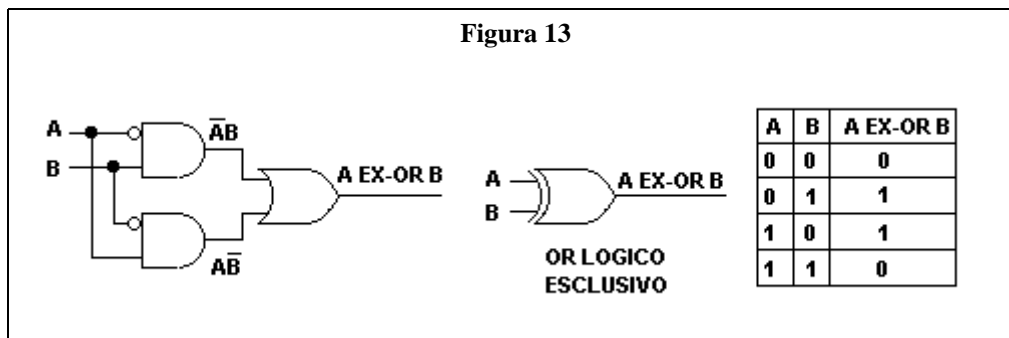



Figura 13

Possiamo dire quindi che se entrambi gli ingressi sono a livello logico **1**, in uscita otterremo un livello logico **0**; la Figura 13 mostra (a sinistra) una delle possibili realizzazioni dell'**EX-OR** attraverso le porte fondamentali **OR**, **AND**, **NOT**.

5.9.2 Porta NOR Esclusivo

In Figura 14 vediamo invece la porta **EX-NOR** chiamata anche **Exclusive NOR** (**NOR Esclusivo**); l'operatore **EX-NOR** viene ottenuto semplicemente negando il segnale prodotto in uscita dall'**EX-OR**:

A	B	A EX-NOR B
0	0	1
0	1	0
1	0	0
1	1	1

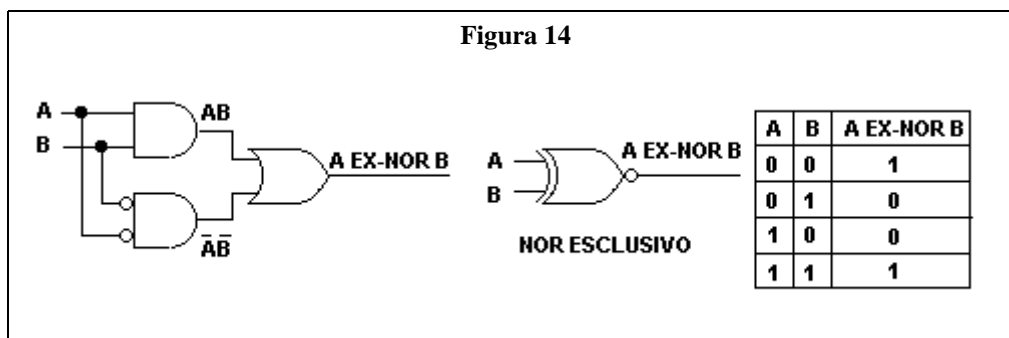



Figura 14

Come si puo' notare dalla tabella della verita', i livelli logici in uscita sono infatti opposti a quelli dell'**EX-OR**; nella Figura 14 a sinistra vediamo anche una possibile realizzazione dell'**EX-NOR** con le porte logiche fondamentali **OR**, **AND**, **NOT**.

5.9.3 Porta NOR

La Figura 15 mostra la porta **NOR** che si ottiene semplicemente negando il risultato prodotto da una porta **OR**; come si vede infatti dalla tabella della verita', i livelli logici che si ottengono in uscita sono esattamente opposti a quelli della porta **OR**:

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

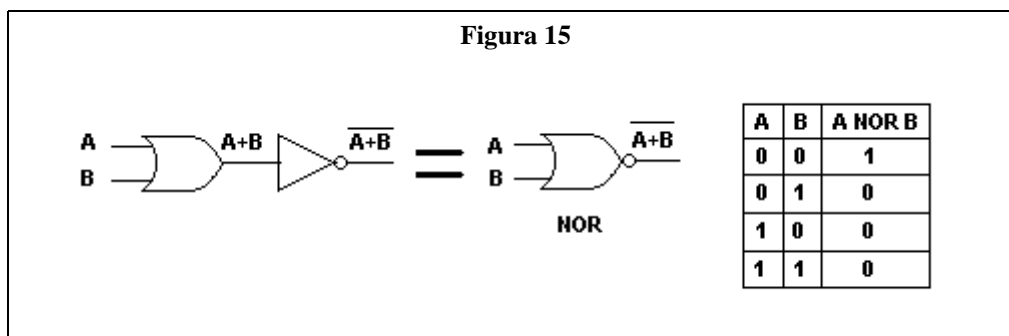



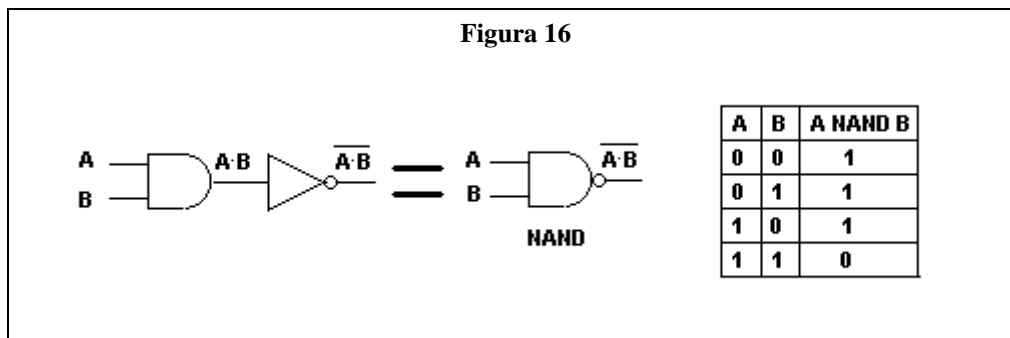
Figura 15

In Figura 15 a sinistra viene mostrata una possibile realizzazione della porta **NOR** con le porte logiche fondamentali; il metodo piu' ovvio da seguire consiste nel ricorrere ad un **OR** e ad un **NOT** collegati in serie.

5.9.4 Porta NAND

La Figura 16 infine mostra la porta **NAND** che si ottiene negando il risultato prodotto da una porta **AND**; anche in questo caso, la tabella della verita' mostra appunto che i livelli logici prodotti dalla porta **NAND** sono esattamente opposti a quelli della porta **AND**:

Figura 16

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

In Figura 16 a sinistra viene mostrata una possibile realizzazione della porta **NAND** con le porte logiche fondamentali; il metodo piu' ovvio da seguire consiste nel ricorrere ad un **AND** e ad un **NOT** collegati in serie.

5.9.5 Porte logiche derivate dalla porta NAND

Le ultime due porte che abbiamo visto (**NOR** e **NAND**) assumono una importanza particolare in quanto, sia l'una che l'altra possono essere utilizzate per realizzare diverse altre porte logiche; questo aspetto e' molto importante in quanto puo' capitare che nella realizzazione pratica di un circuito digitale, venga a mancare qualche porta logica **NOT**, **OR**, **EX-OR**, etc. La soluzione consiste allora nell'utilizzare appunto uno di questi due componenti (**NOR** o **NAND**) per realizzare le porte mancanti; la Figura 17 ad esempio mostra una serie di porte logiche ottenibili a partire dalla porta **NAND**:

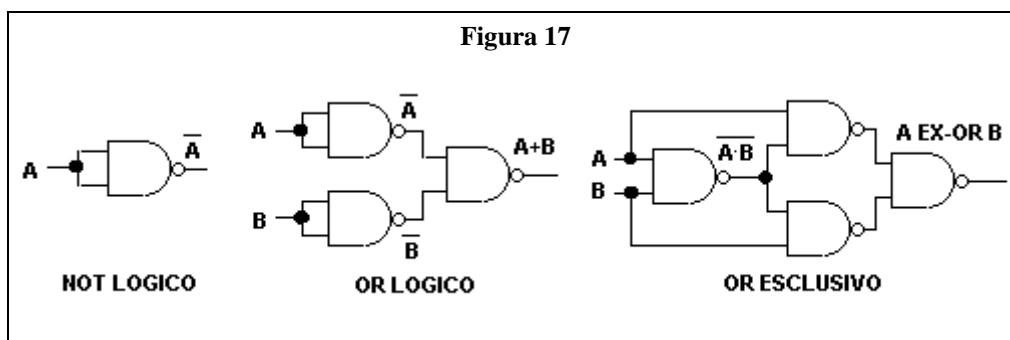



Figura 17

La porta **AND** si puo' ottenere facendo seguire ad un **NAND** il **NOT** che si vede in Figura 17; considerando il fatto che a partire dalle porte **NOT**, **OR**, **AND** si puo' realizzare qualunque altra porta logica, si capisce subito che e' possibile (come spesso avviene in pratica) realizzare un intero circuito digitale utilizzando solo porte **NAND** (o solo porte **NOR**).

In questo capitolo abbiamo appurato quindi che componendo una serie di proposizioni logiche **A**, **B**, **C**, etc, attraverso le porte **NOT**, **OR**, **AND**, **NOR**, etc, si ottiene una cosiddetta **espressione**

booleana che produrra' come risultato un valore binario **0** o **1**; ad esempio:

```
f = A AND (C OR (NOT B))
```

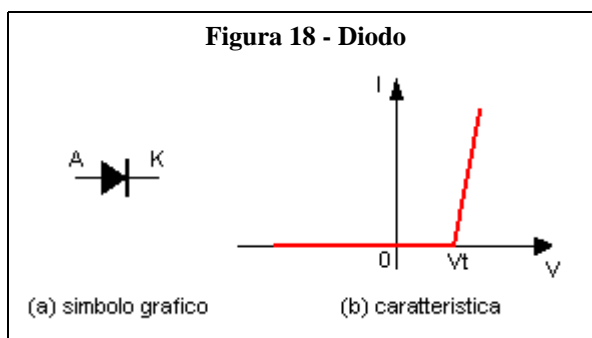
In base alle considerazioni esposte in precedenza, e' evidente che qualsiasi espressione booleana, potra' essere implementata fisicamente attraverso una **R.C.**; nel prossimo capitolo parleremo in particolare delle **R.C.** che ricevono in input dei numeri binari ed effettuano su di essi delle operazioni matematiche producendo in output dei risultati sempre sotto forma di numeri binari.

5.10 Le famiglie logiche

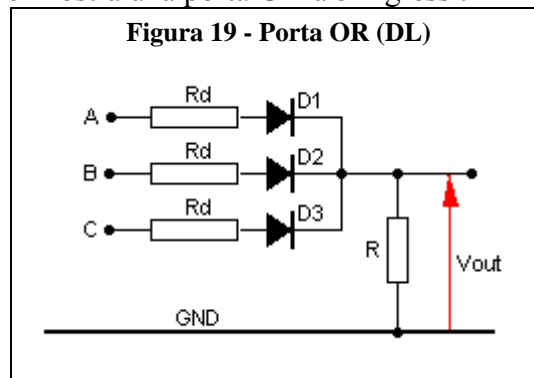
Dopo aver esaminato gli aspetti teorici su cui si basano i circuiti logici del computer, puo' essere interessante analizzare i metodi che vengono impiegati per la realizzazione pratica delle principali porte logiche; nel corso degli anni sono state impiegate a tale scopo diverse tecnologie sempre piu' evolute, che vengono indicate con il nome di **famiglie logiche**. Naturalmente, nel seguito del capitolo questo argomento verra' trattato in modo molto semplificato; per maggiori dettagli si consiglia di consultare un testo di elettronica digitale.

5.10.1 La famiglia DL (Diode Logic)

Storicamente la prima famiglia logica comparsa sul mercato, e' stata la famiglia **DL** o **Diode Logic** (logica a diodi); questo nome deriva dal fatto che questa tecnologia e' basata sull'uso dei **diodi**. La Figura 18 illustra il simbolo grafico del diodo (Figura 18a) e la curva che rappresenta la caratteristica di funzionamento di questo dispositivo elettronico (Figura 18b):

Figura 18 - Diodo

Il terminale **A** del diodo prende il nome di **anodo**, mentre il terminale **K** prende il nome di **catodo**; semplificando al massimo possiamo dire che il diodo e' una sorta di valvola elettronica che si lascia attraversare dalla corrente elettrica solo in un verso (quello indicato dal triangolo nero). Se sul terminale **A** viene applicato un potenziale elettrico maggiore di quello presente sul terminale **K**, si dice che il diodo e' **polarizzato direttamente**; come si nota dalla curva caratteristica di Figura 18b (semiasse positivo **V**), non appena la differenza di potenziale **V** supera la tensione **Vt**, il diodo entra in conduzione comportandosi con buona approssimazione come un interruttore chiuso (**ON**). La tensione **Vt** si chiama **tensione di soglia**, e per i diodi al silicio vale circa **0.5V**.
Se sul terminale **A** viene applicato un potenziale elettrico minore di quello presente sul terminale **K**, si dice che il diodo e' **polarizzato inversamente**; come si nota dalla curva caratteristica di Figura 18b (semiasse negativo **V**), in queste condizioni il diodo e' in **interdizione** e impedisce il passaggio della corrente. Il diodo in interdizione si comporta quindi con buona approssimazione come un interruttore aperto (**OFF**).

Sulla base delle considerazioni appena esposte, possiamo ora analizzare le porte logiche realizzabili in tecnologia **DL**; la Figura 19 mostra una porta **OR** a **3** ingressi:

Figura 19 - Porta OR (DL)

Assumiamo come al solito che la massa (**GND**) sia a potenziale di **0V**, e che il positivo di alimentazione (**+Vcc**) sia a potenziale di **+5V**; questi due valori di tensione rappresentano come al solito il livello logico **0** e il livello logico **1**. Osserviamo che in fase di conduzione un diodo si comporta come un vero e proprio corto circuito, e quindi puo' essere danneggiato dalla corrente intensa che lo attraversa; proprio per questo motivo i diodi vanno usati sempre con una resistenza in serie (**Rd** in Figura 19) che ha lo scopo di limitare la corrente.
Supponiamo ora che i tre ingressi **A**, **B**, **C** siano tutti a potenziale di massa (**0V**), cioe' a livello logico **0**; in queste condizioni i tre diodi non possono condurre (in quanto non sono polarizzati) e quindi si trovano tutti allo stato **OFF**. La corrente in uscita vale **0**, e quindi anche la resistenza **R** (resistenza di **pull-down**) viene attraversata da una corrente **I=0**; in base alla legge di **Ohm** quindi otteniamo:

```
Vout = R x I = R x 0 = 0V = GND
```

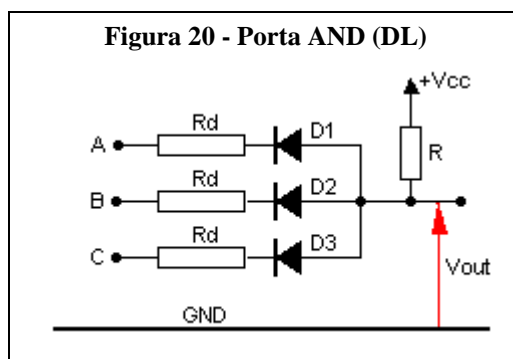
In definitiva, se tutti gli ingressi sono a livello logico basso, anche in uscita (**Vout**) si ottiene un livello logico basso.
Supponiamo ora che almeno uno degli ingressi (ad esempio **A**) si trovi a potenziale **+Vcc=+5V**, cioe' a livello logico alto; in questo caso i due diodi **D2** e **D3** non conducono (**OFF**) in quanto non sono polarizzati, mentre il diodo **D1** entra in conduzione (**ON**) in quanto e' polarizzato direttamente.
Tenendo conto del fatto che la resistenza **Rd** e la resistenza interna del diodo sono in genere trascurabili rispetto a **R**, si ottiene:

```
Vout = +Vcc - Vt = 5 - 0.5 = 4.5V
```

Si tenga presente che i circuiti logici hanno necessariamente una certa tolleranza, per cui riconoscono come livello logico **0** una tensione prossima a **0V** (ad esempio tra **-1V** e **+1V**), e riconoscono come livello logico **1** una tensione prossima a **+5V** (ad esempio tra **+4V** e **+6V**); tutte le tensioni intermedie (ad esempio tra **+1V** e **+4V**) rappresentano invece uno stato di ambiguita' che si presenta solo nell'istante in cui avviene la commutazione da **ON** a **OFF** e viceversa. Possiamo dire quindi che se almeno uno degli ingressi di **Figura 19** e' a livello logico alto, anche l'uscita si portera' a livello logico alto; in definitiva, il circuito di Figura 19 implementa la funzione:

```
f = A OR B OR C
```

Attraverso la tecnologia **DL** e' possibile realizzare anche le porte **AND**; la Figura 20 mostra ad esempio una porta **AND** a **3** ingressi:

Figura 20 - Porta AND (DL)

Supponiamo che almeno uno degli ingressi (ad esempio **A**) si trovi a potenziale di massa **GND=0V**, cioe' a livello logico basso; in queste condizioni il diodo **D1** e' polarizzato direttamente (**ON**), mentre i due diodi **D2** e **D3** hanno entrambi i terminali collegati a **+Vcc** e quindi sono in interdizione (**OFF**). Tutta la corrente **I** che arriva dal positivo di alimentazione, attraversa la resistenza **R** (resistenza di **pull-up**) e si scarica a massa attraverso **D1**; possiamo dire quindi che:

```
Vout = +Vcc - (R x I)
```

Indicando con **Ri** la resistenza interna del diodo, possiamo scrivere:

```
I = (+Vcc - Vt) / (Ri + Rd + R)
```

Abbiamo quindi:

```
Vout = +Vcc - (R x I) = +Vcc - R x (+Vcc - Vt) / (Ri + Rd + R)
```

Ricordando che **Ri** e **Rd** sono trascurabili rispetto a **R**, si ottiene:

```
Vout = +Vcc - R x (+Vcc - Vt) / R = +Vcc - +Vcc + Vt = Vt = +0.5V
```

La tensione in uscita di **0.5V** viene riconosciuta come un livello logico basso; in definitiva, se almeno un ingresso e' a livello logico basso, anche in uscita (**Vout**) si ottiene un livello logico basso.
Supponiamo ora che tutti gli ingressi **A**, **B**, **C** si trovino a potenziale **+Vcc=+5V**, cioe' a livello logico **1**; in questo caso, i tre diodi **D1**, **D2** e **D3**, avendo entrambi i terminali collegati a **+Vcc**, sono tutti in interdizione (**OFF**). Dal positivo di alimentazione non arriva nessuna corrente, e quindi in **R** non si verifica nessuna caduta di tensione; si ottiene quindi:

```
Vout = +Vcc - (R x I) = +Vcc - 0 = +Vcc = +5V
```

Possiamo dire quindi che se tutti gli ingressi sono a livello logico alto, anche in uscita si ottiene un livello logico alto; in definitiva, il circuito di Figura 20 implementa la funzione:

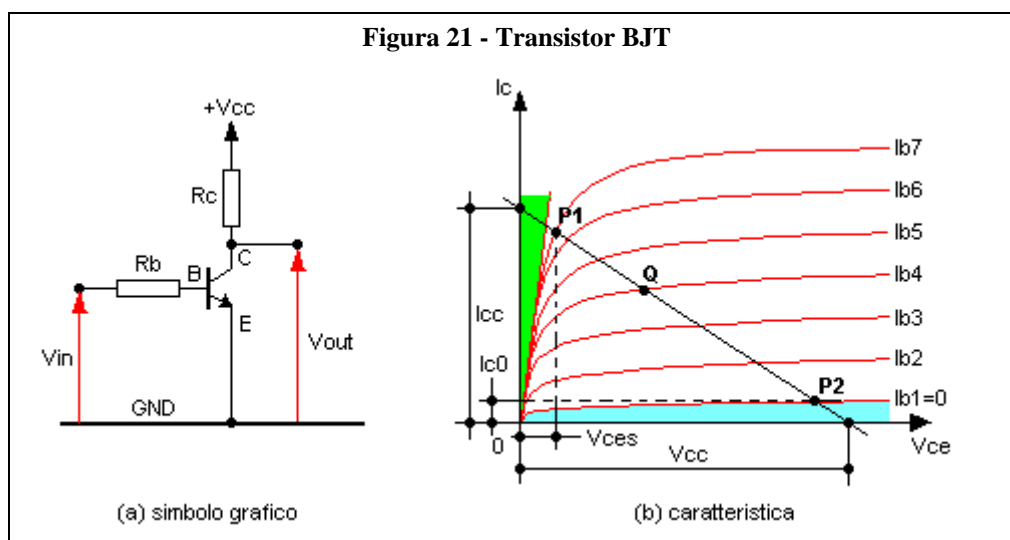
```
f = A AND B AND C
```

Le considerazioni appena esposte ci permettono di dedurre una serie di pregi e difetti della famiglia di porte logiche **DL**.
Osserviamo innanzi tutto che la curva caratteristica di Figura 18b dimostra come i diodi abbiano una elevata velocita' di risposta, che si traduce in un bassissimo **tempo di propagazione** (tempo che intercorre tra l'arrivo in ingresso dei segnali da elaborare e la comparsa in uscita del segnale elaborato); come si puo' facilmente intuire, questo e' uno dei parametri piu' importanti per una porta logica.
Un grave difetto della famiglia **DL** e' costituito dalla elevata caduta di tensione provocata dalle porte **OR**; abbiamo visto infatti che applicando su un ingresso di una porta **OR** un livello logico alto (**+5V**), si ottiene in uscita un livello logico alto di **+4.5V**. Tutto cio' significa che in presenza di diverse porte **OR** collegate in serie, si ottiene una caduta di tensione eccessiva che compromette il funzionamento del circuito logico.
Un secondo importante difetto della famiglia **DL** e' dato dal fatto che con i diodi non e' possibile realizzare porte **NOT**.
Un ulteriore grave difetto e' costituito dal fatto che le caratteristiche costruttive delle porte in tecnologia **DL** rendono praticamente impossibile la miniaturizzazione delle porte stesse; tutto cio' si

traduce in un ingombro eccessivo dei circuiti logici realizzati con i diodi.
Come si puo' constatare, i difetti della famiglia **DL** superano abbondantemente i pregi; proprio per questo motivo questa tecnologia e' stata quasi del tutto abbandonata.

5.10.2 La famiglia TTL (Transistor Transistor Logic)

Con l'invenzione del transistor, il mondo dell'elettronica ha subito una vera e propria rivoluzione; questa rivoluzione ha avuto naturalmente importantissime ripercussioni positive anche sulle tecnologie costruttive delle varie famiglie logiche. In particolare, grazie ai transistor sono nate le famiglie logiche **DTL** (**Diode Transistor Logic**) e la famosissima **TTL** (**Transistor Transistor Logic**); la Figura 21 illustra il simbolo grafico del transistor tipo **BJT** completo di resistenze di polarizzazione (Figura 21a), e la curva che rappresenta la sua caratteristica di funzionamento (Figura 21b):

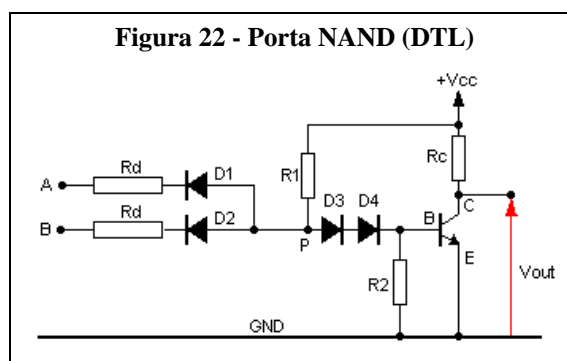
Figura 21 - Transistor BJT

La sigla **BJT** sta per **Bipolar Junction Transistor** (transistor a giunzione bipolare); come possiamo notare in Figura 21a, il **BJT** ha tre terminali chiamati **base** (**B**), **collettore** (**C**) e **emettitore** (**E**).
Semplificando al massimo si puo' dire che il **BJT** e' una sorta di regolatore di corrente; in pratica, regolando da un minimo ad un massimo la corrente **Ib** che giunge alla base, e' possibile regolare da un minimo ad un massimo la corrente **Ic** che transita dal collettore all'emettitore. Da queste considerazioni segue anche il fatto che il **BJT** e' un transistor **pilotato in corrente**.
Analizziamo ora il funzionamento del **BJT** con l'ausilio delle curve caratteristiche di Figura 21b; si tenga presente che queste curve vengono fornite direttamente dal produttore. Supponiamo innanzi tutto di alimentare il circuito di Figura 21a con la tensione **+Vcc=+5V**; riportiamo il valore **+Vcc** sull'asse **Vce** del grafico di Figura 21b. Per la legge di **Ohm**, la massima corrente di collettore sara' quindi:

```
Icc = Vcc / Rc
```

Riportiamo il valore **Icc** sull'asse **Ic** del grafico di Figura 21b; congiungendo questi due punti, si ottiene la cosiddetta **retta di carico** del transistor.
A questo punto, imponendo una determinata corrente di base **Ib**, otteniamo il cosiddetto **punto di lavoro** del transistor, e cioe' il punto di intersezione tra la retta di carico e la curva relativa a **Ib**; in Figura 21b vediamo ad esempio che in corrispondenza della corrente di base **Ib4**, otteniamo il punto di lavoro **Q**. La retta verticale passante per **Q** individua sull'asse **Vce** la **tensione di collettore**, cioe' la tensione **Vout** prelevabile dall'uscita di collettore; la retta orizzontale passante per **Q** individua sull'asse **Ic** la **corrente di collettore**, cioe' la corrente **Ic** che entra dal collettore del transistor.
Si tenga presente che per i **BJT** di piccola potenza, la corrente di base e' dell'ordine di qualche microampere (milionesimi di ampere), mentre la corrente di collettore e' dell'ordine di qualche milliampere (millesimi di ampere); in sostanza, piccolissime variazioni della corrente di base,

producono grandi variazioni della corrente di collettore. Da queste considerazioni si deduce che il **BJT** puo' essere quindi usato in funzione di dispositivo **amplificatore**; osserviamo infatti che applicando ad esempio sulla base un debole segnale radio captato da una antenna, otteniamo in uscita lo stesso segnale radio, ma notevolmente amplificato.
Per la realizzazione delle famiglie logiche, il **BJT** non viene usato come amplificatore, ma bensi' come interruttore elettronico; per capire questa importante caratteristica dei **BJT**, analizziamo ancora la Figura 21b. Applicando sulla base una corrente nulla (**Ib1=0**), si ottiene la minima corrente di collettore possibile; questa corrente viene indicata in Figura 21b con **Ic0**, ed e' anch'essa quasi nulla. In queste condizioni, la caduta di tensione provocata da **Rc** e' molto piccola, per cui il potenziale **+Vcc** si trasferisce quasi interamente sull'uscita di collettore (**Vout**); come si puo' notare in Figura 21b, in corrispondenza di **Ib1=0** il punto di lavoro del **BJT** si porta in posizione **P2**. In sostanza, possiamo dire che per **Ib1=0** si ottiene **Vout** circa uguale a **+Vcc**, e **Ic0** circa uguale a zero; in queste condizioni si dice che il **BJT** e' in stato di **interdizione** (zona celeste), e il suo comportamento equivale con buona approssimazione ad un interruttore aperto (**OFF**) che impedisce il transito della corrente dal collettore all'emettitore.
Applicando ora sulla base una corrente elevatissima come **Ib7**, si ottiene una altrettanto elevata corrente di collettore; come si nota in Figura 21b, questa corrente tende verso il massimo valore possibile **Icc**. In queste condizioni, la caduta di tensione provocata da **Rc** e' molto grande, per cui il potenziale **+Vcc** viene quasi completamente "eroso" dalla stessa caduta di tensione; sull'uscita di collettore sara' quindi possibile prelevare una tensione **Vout** prossima allo zero, indicata in Figura 21b con **Vces**. Come si puo' notare sempre dalla Figura 21b, in corrispondenza di **Ib7** il punto di lavoro del **BJT** si porta in posizione **P1**; applicando sulla base correnti maggiori di **Ib7**, si ottiene una corrente di collettore che cresce in modo insignificante in quanto come si puo' facilmente intuire, il **BJT** e' ormai "saturo". In sostanza, possiamo dire che per **Ib7** si ottiene **Vout** circa uguale a zero (**Vces**), e **Ic** circa uguale a **Icc**; in queste condizioni si dice che il **BJT** e' in stato di **saturazione** (zona verde), e il suo comportamento equivale con buona approssimazione ad un interruttore chiuso (**ON**) che lascia scorrere liberamente la corrente che va' dal collettore all'emettitore.
A questo punto e' facile capire come sia possibile trasformare il **BJT** in un interruttore elettronico; per far svolgere al **BJT** questa funzione, basta portarlo rapidamente dalla saturazione all'interdizione o viceversa (dal punto **P2** al punto **P1** o viceversa). Si tenga presente che costruttivamente il **BJT** e' formato in pratica da due diodi contrapposti, per cui la sua velocita' di risposta sara' elevatissima; come e' stato gia' detto in precedenza, la velocita' di risposta e' un parametro di enorme importanza per le porte logiche.
Dai concetti appena esposti, si deduce anche una ulteriore importante caratteristica del **BJT**; assumiamo come al solito che il potenziale **GND=0V** rappresenti il livello logico **0**, e che il potenziale **+Vcc=+5V** rappresenti il livello logico **1**. Applichiamo ora alla base un potenziale **Vin=+Vcc**; cio' equivale ad applicare in ingresso un livello logico **1**. Dimensionando opportunamente **Rb**, in corrispondenza di **Vin=+Vcc** possiamo far arrivare alla base una corrente maggiore di **Ib7** (Figura 21b); in questo caso sappiamo che il **BJT** si porta in saturazione, per cui in uscita otterremo **Vout** circa uguale a zero. In sostanza, applicando in ingresso un livello logico **1**, otteniamo in uscita (di collettore) un livello logico **0**.
Applichiamo ora alla base un potenziale **Vin=0**; cio' equivale ad applicare in ingresso un livello logico **0**. In corrispondenza di **Vin=0**, arriva alla base una corrente **Ib1=0** (Figura 21b); in questo caso sappiamo che il **BJT** si porta in interdizione, per cui in uscita otterremo **Vout** circa uguale a **+Vcc**. In sostanza, applicando in ingresso un livello logico **0**, otteniamo in uscita (di collettore) un livello logico **1**.
A questo punto abbiamo appurato che il **BJT** con uscita di collettore non e' altro che una semplicissima porta **NOT**; questa porta **NOT** puo' essere impiegata singolarmente, oppure per negare porte **OR**, **AND**, etc, realizzate sempre con i transistor.
Nella sezione **5.9** di questo capitolo abbiamo visto che a partire dalla porta **NAND** (o **NOR**), e'

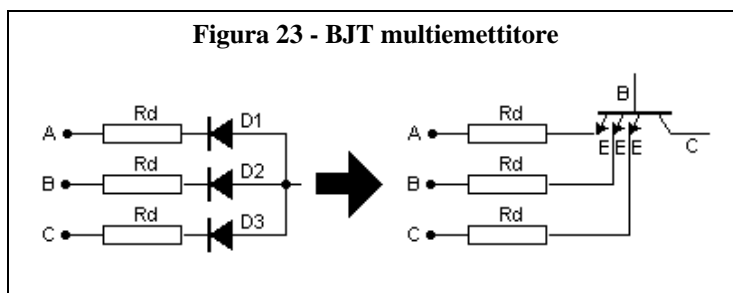
possibile realizzare qualsiasi altra porta logica; vediamo allora come puo' essere realizzata una porta **NAND** mediante l'uso di diodi e transistor (**DTL**). La Figura 22 illustra lo schema di una porta **NAND** in tecnologia **DTL**:

Figura 22 - Porta NAND (DTL)

Supponiamo che almeno uno dei due ingressi (ad esempio **A**) sia collegato a massa (cioe' a livello logico **0**); in questo caso il diodo **D1** e' polarizzato direttamente ed e' quindi in conduzione, mentre il diodo **D2** ha entrambi i terminali a **+Vcc** e quindi non essendo polarizzato e' in interdizione. La corrente che arriva dal positivo di alimentazione si scarica quindi a massa tramite **D1** portando il potenziale del punto **P** quasi a zero; il potenziale del punto **P** viene ulteriormente "eroso" dalle tensioni di soglia dei diodi **D3** e **D4**, e quindi non riesce a portare il **BJT** in conduzione. Il **BJT** e' in interdizione (**OFF**) e quindi sull'uscita di collettore troviamo **Vout** circa uguale a **+Vcc**; in sostanza, se almeno uno dei due ingressi e' a livello logico basso, l'uscita sara' a livello logico alto. Si tenga presente che tra il punto **P** e la base del **BJT** sono presenti due diodi in quanto la caduta di tensione (**Vt**) provocata da un solo diodo potrebbe non essere sufficiente per tenere il transistor in interdizione.

Supponiamo ora che entrambi gli ingressi siano a potenziale **+Vcc** (cioe' a livello logico **1**); in questo caso i due diodi **D1** e **D2** hanno entrambi i terminali a potenziale **+Vcc**, e quindi non essendo polarizzati si trovano in interdizione. La corrente che arriva dal positivo di alimentazione, passa attraverso **D3** e **D4** e si riversa in buona parte sulla base del **BJT** portandolo in conduzione; dimensionando opportunamente i vari componenti del circuito di Figura 22, si puo' fare in modo che la corrente di base sia piu' che sufficiente per saturare il **BJT** (**ON**). In questo modo sull'uscita di collettore troviamo **Vout** circa uguale a zero; in sostanza, se tutti gli ingressi sono a livello logico alto, l'uscita sara' a livello logico basso.

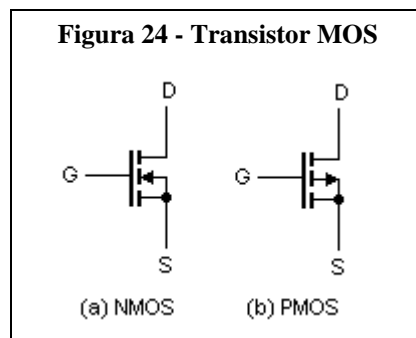
La tabella della verita' del circuito di Figura 22 dimostra quindi che abbiamo a che fare proprio con una porta **NAND**.

L'evoluzione della famiglia **DTL** ha portato alla nascita delle porte logiche in tecnologia **TTL**; come si intuisce dal nome, in questa tecnologia i diodi vengono sostituiti quasi del tutto da transistor. Ricordiamo infatti che di fatto un **BJT** e' formato da due diodi contrapposti; di conseguenza ogni diodo posto all'ingresso di una porta logica **DTL**, puo' essere sostituito dalla giunzione base emettitore di un **BJT**. Invece di utilizzare un **BJT** per ogni ingresso, possiamo servirci di un unico **BJT** dotato di due o piu' emettitori (**BJT multiemettitore**); in questo modo e' possibile ottenere una elevatissima miniaturizzazione delle porte logiche **TTL**. La figura 23 mostra appunto come i tre ingressi di una porta **NAND DTL** possano essere sostituiti da un solo **BJT** a tre emettitori:

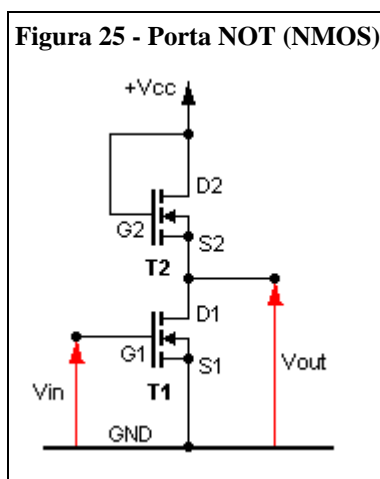
Figura 23 - BJT multiemettitore

La famiglia **TTL** e le altre famiglie da essa derivate, presentano una numerosa serie di vantaggi ai quali si contrappongono pochissimi svantaggi; uno dei vantaggi piu' importanti e' rappresentato dalla elevatissima velocita' di risposta, che nella serie **High Speed** (alta velocita') puo' scendere anche sotto i **6** nanosecondi (**1** nanosecondo = **1** miliardesimo di secondo). Un altro vantaggio che assume un grande rilievo e' dato dalla elevata possibilita' di miniaturizzazione delle porte **TTL** e derivate; in questo modo e' possibile realizzare microcircuiti contenenti al loro interno una enorme quantita' di porte logiche.
Lo svantaggio principale della famiglia **TTL** e' dato dalle elevate correnti di lavoro e dalla conseguente sensibile potenza dissipata (diversi milliwatt per porta); in presenza di elevate correnti di lavoro, si creano sensibili disturbi interni in fase di commutazione da **ON** a **OFF** e viceversa, che aumentano notevolmente il rischio di commutazioni indesiderate. Nella famiglia **TTL** e' necessario quindi tenere adeguatamente separati i livelli logici **0** e **1**; come abbiamo visto, viene utilizzata la tensione di **0V** per il livello logico **0**, mentre per il livello logico **1** non e' consigliabile scendere sotto i **+5V**.
In ogni caso, i vantaggi delle porte **TTL** superano abbondantemente gli svantaggi; tanto e' vero che la famiglia **TTL** presenta una enorme diffusione sul mercato.

5.10.3 La famiglia CMOS

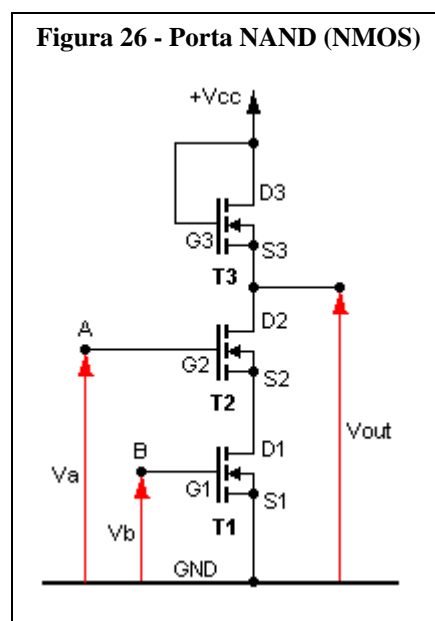
Come abbiamo appena visto, il tallone di Achille della famiglia **TTL** (e derivate), e' rappresentato dal sensibile valore che assume la potenza dissipata; per risolvere questo problema sarebbe necessario ovviamente ridurre l'intensita' della corrente di lavoro. E' stata realizzata a tale proposito una serie **Low Power** (bassa potenza) della famiglia **TTL** che permette di ridurre la potenza dissipata sino a **1** milliwatt per porta; purtroppo pero' questa tecnologia presenta l'inconveniente di un costo di produzione relativamente elevato.
La situazione e' cambiata in modo radicale con la comparsa sul mercato del **Field Effect Transistor** o **FET** (transistor ad effetto di campo); in relazione alla tecnologia costruttiva basata sui semiconduttori all'ossido di metallo, il **FET** viene anche chiamato **MOSFET** o semplicemente **MOS** (**Metal Oxide Semiconductor**). La Figura 24 illustra il simbolo grafico di un **MOS** a canale **N** (**NMOS**) e di un **MOS** a canale **P** (**PMOS**); le curve caratteristiche di questi transistor sono del tutto simili a quelle del **BJT**:

Figura 24 - Transistor MOS

Come possiamo notare, il **MOS** ha un terminale **G** chiamato **gate** (cancello), un terminale **D** chiamato **drain** (canale di scarico) e un terminale **S** chiamato **source** (sorgente); la differenza fondamentale che esiste tra **BJT** e **MOS** e' data dal fatto che il **BJT** come gia' sappiamo e' pilotato in corrente, mentre il **MOS** e' **pilotato in tensione**. Il gate infatti e' elettricamente isolato dal drain e dal source grazie ad uno strato di biossido di silicio (**SiO2**, da cui deriva il nome **MOS**); regolando da un minimo ad un massimo la tensione applicata al gate, e' possibile regolare la larghezza del canale che collega il drain al source. Attraverso la regolazione della larghezza del canale si ottiene anche la regolazione della sua resistenza elettrica; possiamo dire quindi che regolando da un minimo ad un massimo la tensione applicata al gate, otteniamo la regolazione da un minimo ad un massimo della corrente che attraversa il canale.
L'aspetto veramente importante e' dato dal fatto che quando vengono impiegati nei circuiti logici, i **MOS** possono lavorare con correnti spesso irrisorie; mediamente le porte logiche realizzate con i **MOS** presentano una potenza dissipata pari a frazioni di milliwatt. Tutto cio' rende possibile un grado di miniaturizzazione estremamente elevato; vengono prodotti infatti microcircuiti contenenti al loro interno milioni di transistor **MOS**!
Un'altra importante caratteristica dei **MOS** e' rappresentata dalla elevata tensione di soglia **Vt** che vale circa **+1.5V**; cio' si traduce in una notevole immunita' ai disturbi elettrici, con la conseguente possibilita' di ridurre il valore di tensione associato al livello logico **1**. Vengono prodotte ad esempio porte logiche basate sui **MOS**, che si servono di una tensione **+Vcc** di appena **+3V** per il livello logico **1**.

Analizziamo ora il comportamento di un **NMOS** utilizzato come interruttore elettronico.
Se la tensione **Vgs** applicata tra gate e source e' inferiore alla tensione di soglia **Vt**, allora il canale tra drain e source e' chiuso, e il **NMOS** e' in interdizione (**OFF**); il canale chiuso impedisce il passaggio della corrente, per cui **Ids=0**.
Se la tensione **Vgs** applicata tra gate e source e' superiore alla tensione di soglia **Vt**, allora il canale tra drain e source e' aperto, e il **NMOS** e' in conduzione (**ON**); nel canale circola una corrente **Ids** maggiore di zero che cresce al crescere di **Vgs** (sino alla saturazione).
Nel caso del **PMOS** la situazione e' opposta.
Se la tensione **Vgs** applicata tra gate e source e' superiore alla tensione di soglia **Vt**, allora il canale tra drain e source e' chiuso, e il **PMOS** e' in interdizione (**OFF**); il canale chiuso impedisce il passaggio della corrente, per cui **Ids=0**.
Se la tensione **Vgs** applicata tra gate e source e' inferiore alla tensione di soglia **Vt**, allora il canale tra drain e source e' aperto, e il **PMOS** e' in conduzione (**ON**); nel canale circola una corrente **Ids** maggiore di zero che cresce al diminuire di **Vgs** (sino alla saturazione).
Sulla base di queste considerazioni possiamo analizzare alcune porte logiche realizzabili con i **MOS**; la Figura 25 illustra ad esempio una porta **NOT** realizzata con i **NMOS**:

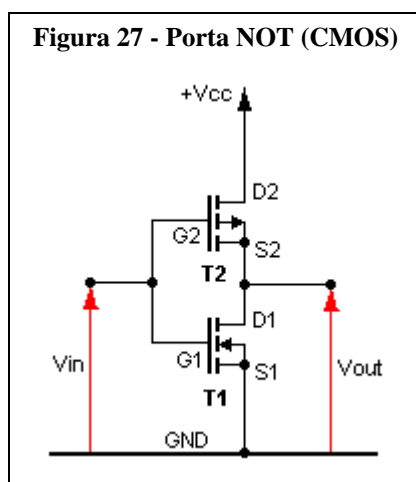
Figura 25 - Porta NOT (NMOS)

In teoria tra il positivo di alimentazione **+Vcc** e il transistor **T1** ci dovrebbe essere una resistenza di polarizzazione; si preferisce pero' utilizzare per tale scopo la resistenza interna di un secondo **NMOS** (**T2**). In questo modo e' possibile ottenere una miniaturizzazione molto piu' spinta della porta logica; osserviamo infatti che un **MOS** e' molto piu' piccolo e miniaturizzabile di una resistenza elettrica.
Supponiamo ora di portare l'ingresso a potenziale di massa (**Vin=0**); cio' equivale ad applicare in ingresso un livello logico zero. Dalla Figura 25 si rileva che **T2** dovrebbe essere teoricamente in conduzione; **T1** pero' e' in interdizione (perche' ha il gate a potenziale zero), e impedisce anche a **T2** di condurre. In Figura 25 notiamo che **G2** e' allo stesso potenziale (**+Vcc**) di **D2**, per cui ricaviamo:
`Vout = Vcc - Vds2 = Vcc - Vgs2`
Siccome **T2** e' costretto all'interdizione, la sua tensione **Vgs2** e' sicuramente inferiore alla sua tensione di soglia **Vt2**; il transistor **T2** puo' essere progettato in modo da avere **Vt2** prossima allo zero, per cui anche **Vgs2** sara' praticamente zero. Otteniamo quindi:
`Vout = Vcc - Vgs2 = Vcc - 0 = Vcc`
Applicando quindi in ingresso un potenziale **Vin=0** (livello logico basso), otteniamo in uscita un potenziale **Vout=+Vcc** (livello logico alto).
Portiamo ora l'ingresso a potenziale **Vin=+Vcc**; cio' equivale ad applicare in ingresso un livello logico **1**. **T1** passa sicuramente in conduzione trascinando in conduzione anche **T2**; la caduta di tensione si ripartisce tra **T1** e **T2**, per cui possiamo scrivere:
`Vcc = Vds1 + Vds2 = Vout + Vds2`
Da questa relazione si ricava:
`Vout = Vcc - Vds2`
Siccome **T2** lavora come carico resistivo, possiamo fare in modo che la sua resistenza interna sia talmente alta da "erodere" quasi tutta la **+Vcc**; in questo modo si ottiene:
`Vout = Vcc - Vds2 = Vcc - Vcc = 0`
Applicando quindi in ingresso un potenziale **Vin=+Vcc** (livello logico alto), otteniamo in uscita un potenziale **Vout=0** (livello logico basso). In definitiva possiamo dire che il circuito di Figura 25 rappresenta una porta **NOT** in tecnologia **NMOS**.

Partendo dalla porta **NOT** e' possibile costruire facilmente una porta **NAND** che come gia' sappiamo, puo' essere utilizzata per ottenere qualsiasi altra porta logica; la Figura 26 mostra proprio una porta **NAND** in tecnologia **NMOS**:

Figura 26 - Porta NAND (NMOS)

Supponiamo che almeno un ingresso (ad esempio **A**) sia a livello logico basso (**Va=0** e **Vb=+Vcc**); in questo caso **T2** e' in interdizione, e impedisce di condurre anche a **T1** (che e' in serie a **T2**). In base a quanto abbiamo visto prima in relazione alla porta **NOT**, l'uscita si portera' a potenziale **+Vcc**; possiamo dire quindi che se almeno un ingresso e' a livello logico basso, l'uscita sara' a livello logico alto.
Supponiamo ora che entrambi gli ingressi siano a livello logico alto (**Va=+Vcc** e **Vb=+Vcc**); in questo caso **T1** e **T2** sono entrambi in conduzione. In base a quanto abbiamo visto prima in relazione alla porta **NOT**, l'uscita si portera' a potenziale **0**; possiamo dire quindi che se tutti gli ingressi sono a livello logico alto, l'uscita sara' a livello logico basso. Abbiamo appurato quindi che il circuito di Figura 26 rappresenta una porta **NAND** in tecnologia **NMOS**.

I transistor **NMOS** e **PMOS** presentano una serie di vantaggi veramente notevoli; oltre alla piccolissima potenza dissipata, bisogna considerare anche una elevatissima possibilita' di miniaturizzazione e un tempo di propagazione molto basso (tra **10** e **20** nanosecondi). Purtroppo pero' questi transistor presentano anche un difetto abbastanza serio; questo difetto e' dato dal fatto che il tempo necessario per la commutazione da **OFF** a **ON** (fronte di salita) e' notevolmente differente dal tempo necessario per la commutazione da **ON** a **OFF** (fronte di discesa).
Per risolvere questo problema si parte dal fatto che come abbiamo visto prima, un **NMOS** ha un comportamento opposto (o per meglio dire complementare) rispetto ad un **PMOS**; la soluzione consiste allora nell'utilizzare una coppia di **MOS** formata da un **NMOS** e da un **PMOS**. In questo modo, il comportamento del **NMOS** viene compensato dal comportamento del **PMOS** e viceversa; la nuova famiglia logica cosi' ottenuta viene chiamata **CMOS** (**Complementary MOS**).

Analizziamo in Figura 27 una porta **NOT** realizzata in tecnologia **CMOS**:

Figura 27 - Porta NOT (CMOS)

Osserviamo subito che portando l'ingresso **Vin** a livello logico alto (**+Vcc**), **T1** (**NMOS**) si porta in conduzione, mentre **T2** (**PMOS**) si porta in interdizione; siccome pero' i due **MOS** sono in serie, **T2** costringera' anche **T1** all'interdizione, e tra i due transistor circolera' quindi una corrente nulla.
Consideriamo ora le curve caratteristiche di Figura 21, dove per i **MOS** bisogna sostituire **Vce** con **Vds**, **Ic** con **Ids** e le curve **Ib** con le curve **Vgs**; da queste curve si puo' notare che per **Ids1=0** (**Ic=0**) si ottiene **Vds1=0** (**Vce=0**), e quindi possiamo scrivere:

```
Vout = Vds1 = 0
```

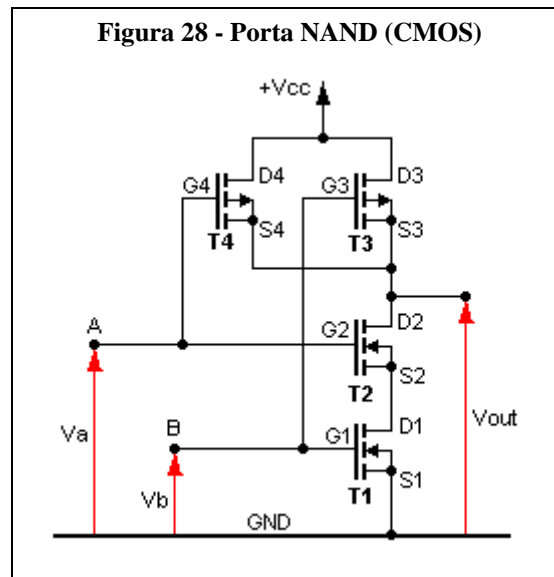
Possiamo dire quindi che se l'ingresso **Vin** e' a livello logico alto, l'uscita **Vout** sara' a livello logico basso.
Portando ora l'ingresso **Vin** a livello logico basso (**0**), **T1** si porta in interdizione, mentre **T2** si porta in conduzione; in questo caso e' **T1** che costringe anche **T2** all'interdizione, e tra i due transistor circolera' quindi una corrente nulla.

Possiamo scrivere allora:
```
Vout = Vcc - Vds2
```

In analogia con il caso precedente, possiamo dire che per **Ids2=0** si ottiene **Vds2=0**; di conseguenza possiamo scrivere:
```
Vout = Vcc - Vds2 = Vcc - 0 = Vcc
```

Possiamo dire quindi che se l'ingresso **Vin** e' a livello logico basso, l'uscita **Vout** sara' a livello logico alto; a questo punto abbiamo appurato che il circuito di Figura 27 rappresenta una porta **NOT** in tecnologia **CMOS**.

Partendo dalla porta **NOT** e' possibile costruire facilmente una porta **NAND** che puo' essere utilizzata per ottenere qualsiasi altra porta logica; la Figura 28 mostra proprio una porta **NAND** in tecnologia **CMOS**:

Figura 28 - Porta NAND (CMOS)

Supponiamo che almeno uno dei due ingressi (ad esempio **A**) sia a livello logico basso (**Va=0** e **Vb=+Vcc**); in questo caso **T2** e' in interdizione, e impedisce di condurre anche a **T1** (che e' in serie a **T2**). Dalla Figura 28 si rileva che il **PMOS T4** ha il gate a livello logico basso, per cui e' in conduzione, mentre il **PMOS T4** ha il gate a livello logico alto, per cui e' interdizione; l'interdizione di **T1** e **T2** costringe all'interdizione anche **T4**, portando **Vds3** e **Vds4** a zero. Possiamo scrivere quindi:
```
Vout = Vcc - Vds3 = Vcc - Vds4 = Vcc - 0 = Vcc
```
Possiamo dire quindi che se almeno un ingresso e' a livello logico basso, l'uscita sara' a livello logico alto.
Supponiamo ora che entrambi gli ingressi siano a livello logico alto (**Va=+Vcc** e **Vb=+Vcc**); in questo caso **T1** e **T2** sono entrambi in conduzione. Dalla Figura 28 si rileva che il **PMOS T3** e il **PMOS T4** hanno entrambi il gate a livello logico alto, per cui sono in interdizione; **T1** e **T2** essendo in conduzione cortocircuitano a massa il potenziale **Vout** per cui si ottiene:
```
Vout = 0
```
Possiamo dire quindi che se entrambi gli ingressi sono a livello logico alto, l'uscita sara' a livello logico basso; abbiamo appurato cosi' che il circuito di Figura 28 rappresenta una porta **NAND** in tecnologia **CMOS**.

La presenza nelle porte **CMOS** di un maggior numero di transistor rispetto alle porte **NMOS** e **PMOS**, aumenta i tempi di propagazione che possono ammontare a diverse decine di nanosecondi; si puo' dire che questo e' l'unico vero difetto delle porte **CMOS**. Per ovviare a questo problema, anche per le porte **CMOS** viene realizzata una apposita serie **High Speed** che permette di raggiungere tempi di propagazione pari a quelli delle porte **TTL**; queste considerazioni unite a cio' che e' stato detto in precedenza, giustificano il fatto che la tecnologia **CMOS** e' quella su cui sta puntando maggiormente l'industria dell'hardware.

Capitolo 6 - Reti combinatorie per funzioni matematiche

Nel Capitolo 4 abbiamo visto che dal punto di vista degli esseri umani, elaborare delle informazioni codificate sotto forma di numeri binari significa eseguire su questi numeri una serie di operazioni matematiche che devono produrre dei risultati sempre in formato binario; in base allora a quanto e' stato esposto nel Capitolo 5, possiamo dire che dal punto di vista del computer, elaborare informazioni codificate sotto forma di segnali logici significa eseguire su questi segnali una serie di operazioni che devono produrre dei risultati sempre sotto forma di segnali logici. In questo capitolo vengono analizzate una serie di **R.C.** che permettono alla **CPU** di eseguire elaborazioni sui segnali logici; per noi esseri umani queste elaborazioni effettuate dal computer assumono proprio l'aspetto di operazioni matematiche eseguite sui numeri binari.

6.1 Circuiti Multiplexer e Demultiplexer

Prima di illustrare le **R.C.** per le funzioni matematiche, analizziamo due circuiti che in elettronica digitale trovano un impiego veramente massiccio; si tratta dei circuiti chiamati **Multiplexer** e **Demultiplexer**.
Il **Multiplexer** o **MUX** rappresenta un vero e proprio commutatore elettronico; questo circuito infatti riceve in ingresso **n** segnali logici, ed e' in grado di selezionare in uscita uno solo di essi. Osserviamo subito che per selezionare una tra **n** possibili linee in ingresso, abbiamo bisogno di **m** linee di selezione con:
$2^m = n$
Da questa relazione si ricava:
$\log_2 2^m = \log_2 n$, e quindi $m = \log_2 n$
Nel caso ad esempio di un **MUX** a **n=4** ingressi, abbiamo bisogno di **m=2** linee di selezione; infatti, con **2** bit possiamo gestire $\mathbf{2^2=4}$ configurazioni diverse (**4** numeri binari differenti).
Se colleghiamo ora i **4** ingressi del **MUX** a **4** porte **AND**, grazie alle **2** linee di selezione possiamo abilitare solo una delle porte **AND**; di conseguenza, la porta **AND** abilitata trasferisce in uscita l'ingresso ad essa collegato. Le **4** uscite delle porte **AND** vengono collegate ad una porta **OR** a **4** ingressi per ottenere il segnale che abbiamo selezionato; la Figura 1 illustra la struttura della **R.C.** che implementa un **MUX** a **4** ingressi e una uscita (chiamato anche **MUX** da **4** a **1**):

Figura 1

L'ingresso **E** (**Enable**) permette di attivare o disattivare l'intero **MUX**; portando l'ingresso **E** a livello logico **1**, in uscita dal **NOT** otteniamo un livello logico **0** che disattiva tutte le **4** porte **AND**. In questo modo, l'uscita **Y** del **MUX** si trova sempre a livello logico basso indipendentemente dai **4** segnali in ingresso; se invece **E=0**, il **MUX** viene attivato, ed e' in grado di selezionare uno solo tra i **4** ingressi **a0**, **a1**, **a2**, **a3**.
Le **2** linee **S0** e **S1** rappresentano gli ingressi di selezione; ponendo ora **E=0** (**MUX** attivo), si puo' constatare dalla Figura 1 che:

***** Se **S0 = 0** e **S1 = 0** si ottiene **Y = a0**.
***** Se **S0 = 1** e **S1 = 0** si ottiene **Y = a1**.
***** Se **S0 = 0** e **S1 = 1** si ottiene **Y = a2**.
***** Se **S0 = 1** e **S1 = 1** si ottiene **Y = a3**.

Il compito opposto rispetto al caso del **MUX** viene svolto da un altro circuito chiamato **Demultiplexer** o **DMUX**; il **DMUX** riceve in ingresso un unico segnale logico, ed e' in grado di trasferirlo su una sola tra le **n** linee di uscita. Come al solito, per la selezione delle **n** linee di uscita abbiamo bisogno di:
$m = \log_2 n$ linee di selezione.
Attraverso queste **m** linee di selezione possiamo attivare solo una delle **n** linee di uscita, disattivando tutte le altre **n-1**; si presenta allora il problema di stabilire il livello logico da assegnare alle **n-1** linee di uscita disabilitate. E' chiaro che se vogliamo trasferire sull'uscita abilitata un segnale a livello logico **1**, dobbiamo portare a livello logico **0** tutte le **n-1** uscite disabilitate; viceversa, se vogliamo trasferire sull'uscita abilitata un segnale a livello logico **0**, dobbiamo portare a livello logico **1** tutte le **n-1** uscite disabilitate.
Per ottenere queste caratteristiche, gli **m** ingressi di selezione agiscono su **n** porte **NAND** ciascuna delle quali fornisce una delle **n** uscite; la Figura 2 illustra la struttura della **R.C.** che implementa un **DMUX** a **1** ingresso e **4** uscite (chiamato anche **DMUX** da **1** a **4**):

Figura 2

In questo circuito, **S0** e **S1** sono gli ingressi di selezione, **D** e' l'unico ingresso per i dati, mentre **E** e' come al solito l'ingresso di abilitazione; le **4** uscite del **DMUX** sono **Y0**, **Y1**, **Y2** e **Y3**.
Il **DMUX** di Figura 2 permette di trasferire su una delle **4** uscite un segnale a livello logico **0**; di conseguenza, le **3** uscite disabilitate si portano a livello logico **1**. Per ottenere questo trasferimento bisogna porre **E=0** e **D=1**; in queste condizioni possiamo constatare che:

***** Se **S0 = 0** e **S1 = 0** si ottiene **Y0 = 0**, **Y1 = 1**, **Y2 = 1**, **Y3 = 1**.
***** Se **S0 = 1** e **S1 = 0** si ottiene **Y0 = 1**, **Y1 = 0**, **Y2 = 1**, **Y3 = 1**.
***** Se **S0 = 0** e **S1 = 1** si ottiene **Y0 = 1**, **Y1 = 1**, **Y2 = 0**, **Y3 = 1**.
***** Se **S0 = 1** e **S1 = 1** si ottiene **Y0 = 1**, **Y1 = 1**, **Y2 = 1**, **Y3 = 0**.

6.2 Comparazione tra numeri binari

Tra le istruzioni piu' importanti messe a disposizione dai linguaggi di programmazione di alto livello, troviamo sicuramente quelle denominate **istruzioni per il controllo del flusso**, cosi' chiamate perche' consentono ad un programma di prendere delle decisioni in base al risultato di una operazione appena eseguita; si possono citare ad esempio istruzioni come **IF**, **THEN**, **ELSE**, **DO WHILE**, **REPEAT UNTIL**, etc. Dal punto di vista della **CPU** tutte queste istruzioni si traducono in una serie di confronti tra numeri binari; proprio per questo motivo, tutte le **CPU** sono dotate di apposite **R.C.** che permettono di effettuare comparazioni tra numeri binari.
Supponiamo ad esempio di avere due numeri binari **A** e **B** a **8** bit che rappresentano i codici ASCII di due lettere dell'alfabeto; consideriamo ora un programma che elabora questi due numeri attraverso le pseudo-istruzioni mostrate in Figura 3:

Figura 3

```
SE (A e' uguale a B)
   esegui la Procedura 1
ALTRIMENTI
   esegui la Procedura 2
```

Come si puo' notare, questa porzione del programma effettua una scelta basata sul risultato del confronto tra **A** e **B**; se **A** e' uguale a **B**, l'esecuzione salta al blocco di istruzioni denominato **Procedura 1**, mentre in caso contrario (**A** diverso da **B**), l'esecuzione salta al blocco di istruzioni denominato **Procedura 2**.
Per comparare **A** con **B**, abbiamo bisogno di una **funzione booleana** che riceve in input i due numeri binari da confrontare e restituisce in output un risultato espresso sempre in forma binaria; possiamo usare ad esempio il valore **0** per indicare che i due numeri sono diversi tra loro, e il valore **1** per indicare che i due numeri sono uguali. Una volta scritta la funzione booleana, si puo' passare alla sua realizzazione pratica tramite una **R.C.**.
Nel precedente capitolo, abbiamo visto che una porta **EX-OR** a due ingressi restituisce **0** se i due livelli logici in ingresso sono uguali tra loro, e restituisce invece **1** in caso contrario; analogamente, una porta **EX-NOR** a due ingressi restituisce **1** se i due livelli logici in ingresso sono uguali tra loro, e restituisce invece **0** in caso contrario. Le porte **EX-OR** oppure le porte **EX-NOR** appaiono quindi particolarmente adatte a risolvere il nostro problema; per capire come si debba effettuare il confronto tra due numeri binari, bisogna fare innanzi tutto alcune considerazioni importanti:

***** Il confronto ha senso solo se i due numeri binari hanno lo stesso numero di bit.
***** Due numeri binari sono uguali quando hanno i bit corrispondenti (di pari posizione) uguali tra loro; quindi il bit in posizione **0** del numero **A** deve essere uguale al bit in posizione **0** del numero **B**, il bit in posizione **1** del numero **A** deve essere uguale al bit in posizione **1** del numero **B** e cosi' via.

In base a queste considerazioni, vediamo subito come si deve procedere utilizzando ad esempio le porte **EX-NOR** che restituiscono **1** se i due livelli logici in ingresso sono uguali tra loro, e **0** se sono diversi; utilizziamo una porta **EX-NOR** per ogni coppia di bit (corrispondenti) da confrontare. La prima porta confronta il bit in posizione **0** del primo numero con il bit in posizione **0** del secondo

numero; la seconda porta confronta il bit in posizione **1** del primo numero con il bit in posizione **1** del secondo numero e cosi' via. Ogni porta **EX-NOR** produce un risultato che vale **1** se i due bit sono uguali tra loro, e vale invece **0** in caso contrario; i due numeri binari sono uguali tra loro solo quando tutte le porte **EX-NOR** producono **1** come risultato, cioe' quando tutti i bit corrispondenti dei due numeri sono uguali tra loro. Se almeno una porta **EX-NOR** produce **0** come risultato, i due numeri binari sono diversi tra loro.
Per sapere se tutte le porte **EX-NOR** hanno prodotto **1**, basta confrontare con un **AND** tutte le loro uscite; l'**AND** ci restituisce **1** solo se tutti i suoi ingressi valgono **1**, e ci restituisce **0** se anche un solo ingresso vale **0**. Facendo riferimento per semplicita' a numeri a **4** bit (nibble), la nostra funzione booleana sara' allora:

```
Y = (A0 EX-NOR B0) AND (A1 EX-NOR B1) AND (A2 EX-NOR B2) AND (A3 EX-NOR B3)
```

dove **A0**, **A1**, **A2**, **A3** rappresentano i **4** bit del primo numero e **B0**, **B1**, **B2**, **B3** rappresentano i **4** bit del secondo numero.
La Figura 4 mostra la **R.C** che implementa in pratica questa funzione:

Figura 4

Comparatore a 4 bit

Si capisce subito che il numero di porte **EX-NOR** da utilizzare e' pari al numero di coppie di bit da confrontare; per confrontare ad esempio due numeri a **8** bit (**8** coppie di bit) occorreranno **8** porte **EX-NOR**. Le uscite di tutte le porte **EX-NOR** impiegate, vanno collegate agli ingressi di una porta **AND** che restituisce **1** solo quando tutti i suoi ingressi valgono **1**; ma questo accade solo quando tutti i confronti hanno prodotto **1** come risultato (bit corrispondenti uguali tra loro). In definitiva, la **R.C.** di Figura 4 fornisce in output il livello logico **1** se i due numeri da confrontare sono uguali tra loro, e il livello logico **0** in caso contrario.
Come si puo' notare, ogni porta **EX-NOR** confronta un bit del primo numero con il bit che nel secondo numero occupa la stessa posizione; questo tipo di confronto viene definito **bit a bit** (in inglese **bitwise**).
Tutti i concetti appena esposti, si estendono immediatamente al confronto tra numeri binari di qualunque dimensione; se vogliamo confrontare numeri binari a **16** bit, ci occorreranno **16** porte **EX-NOR** e una porta **AND** a **16** ingressi; se vogliamo confrontare numeri binari a **32** bit ci occorreranno **32** porte **EX-NOR** e una porta **AND** a **32** ingressi e cosi' via. Queste considerazioni comunque sono solamente teoriche perche' in realta', nella realizzazione pratica dei comparatori (e di altre **R.C.**) si segue un'altra strada; puo' capitare infatti che in commercio si trovino solo i comparatori a **4** bit illustrati in Figura 4. Come si puo' facilmente intuire, in questo caso basta utilizzare uno o piu' comparatori a **4** bit collegati in parallelo. Se ad esempio vogliamo confrontare numeri a **16** bit, utilizzeremo **4** comparatori a **4** bit in parallelo (**4x4=16**); se invece vogliamo confrontare numeri a **32** bit, utilizzeremo **8** comparatori a **4** bit in parallelo (**8x4=32**) e cosi' via. Ogni comparatore ha una sola uscita e tutte queste uscite vanno collegate agli ingressi di una porta **AND** che produrra' il risultato finale. Si ricorda che le porte **AND** con piu' di due ingressi,

producono in uscita un livello logico alto solo se tutti i livelli logici in ingresso sono alti; in caso contrario verra' prodotto un livello logico basso.

Un caso che si presenta molto piu' di frequente consiste nel voler conoscere la relazione esatta che esiste tra due numeri binari; dati cioe' i due numeri binari **A** e **B**, vogliamo sapere se **A** e' minore di **B**, o se **A** e' uguale a **B**, o se **A** e' maggiore di **B**. In Figura 5 vediamo un esempio relativo sempre al confronto tra nibble:

Figura 5

Prima di tutto osserviamo in Figura 5a la struttura elementare del circuito che confronta il bit **Ai** del numero **A** con il corrispondente bit **Bi** del numero **B**; come si puo' facilmente constatare, in base al risultato del confronto solo una delle tre uscite sara' a livello logico alto, mentre le altre due saranno a livello logico basso. In particolare notiamo che la prima uscita (quella piu' in alto) presenta un livello logico alto solo se **Ai** e' maggiore di **Bi** (cioe' **Ai=1** e **Bi=0**); la seconda uscita presenta un livello logico alto solo se **Ai** e' uguale a **Bi** (cioe' **Ai=0** e **Bi=0**, oppure **Ai=1** e **Bi=1**). La terza uscita infine presenta un livello logico alto solo se **Ai** e' minore di **Bi** (cioe' **Ai=0** e **Bi=1**).
Partendo dal circuito elementare di Figura 5a si puo' facilmente ottenere un comparatore completo a **4** bit che ci permette di sapere quale relazione esiste tra il nibble **A** e il nibble **B**; il blocco che rappresenta simbolicamente questo comparatore, viene mostrato in Figura 5b. I comparatori a **4** bit che si trovano in commercio, sono dotati come si puo' notare di **3** uscite indicate con **Y1**, **Y2** e **Y3**; l'uscita **Y1** e' associata alla condizione **A** maggiore di **B**, l'uscita **Y2** e' associata alla condizione **A** uguale a **B** e l'uscita **Y3** e' associata alla condizione **A** minore di **B**.
Possiamo dire quindi che se **A** e' maggiore di **B** si ottiene:
`Y1 = 1, Y2 = 0, Y3 = 0`
Se **A** e' uguale a **B** si ottiene:
`Y1 = 0, Y2 = 1, Y3 = 0`
Se **A** e' minore di **B** si ottiene:
`Y1 = 0, Y2 = 0, Y3 = 1`
La tecnica che si utilizza per il confronto e' molto semplice e si basa su una proprieta' fondamentale dei sistemi di numerazione posizionali; come sappiamo infatti, i numeri rappresentati con questo sistema, sono formati da una sequenza di cifre il cui peso dipende dalla posizione della cifra stessa nel numero. La cifra di peso maggiore e' quella piu' a sinistra e quindi, per confrontare due numeri nel modo piu' rapido possibile, e' necessario partire dal confronto tra le cifre piu' significative dei

due numeri stessi (**MSB** per i numeri binari); ancora una volta bisogna ricordare che l'operazione di confronto ha senso solo se i due numeri hanno lo stesso numero di cifre.
Tornando allora al caso del comparatore a **4** bit di Figura 5b, vediamo come si svolge il confronto tra i due nibble **A** e **B**:

1) Se il bit **A3** e' maggiore del bit **B3** e' inutile continuare perche' sicuramente il nibble **A** e' maggiore del nibble **B**; di conseguenza **Y1** sara' a livello logico **1** mentre le altre due uscite saranno a livello logico **0**.
2) Se il bit **A3** e' minore del bit **B3** e' inutile continuare perche' sicuramente il nibble **A** e' minore del nibble **B**; di conseguenza **Y3** sara' a livello logico **1** mentre le altre due uscite saranno a livello logico **0**.
3) Se il bit **A3** e' uguale al bit **B3** si deve necessariamente passare ai bit in posizione **2** ripetendo i passi **1** e **2** appena descritti; se anche il bit **A2** e' uguale al bit **B2**, si passa ai bit in posizione **1** e cosi' via, sino ad arrivare eventualmente ai bit in posizione **0** (**LSB**). Solo dopo quest'ultimo confronto possiamo dire con certezza se i due numeri sono uguali tra loro e in questo caso avremo **Y2** a livello logico **1** e le altre uscite a livello logico **0**.

Nella Figura 5b si nota anche la presenza di tre connessioni **I1**, **I2** e **I3**; queste connessioni servono per il collegamento in parallelo di due o piu' comparatori quando si ha la necessita' di confrontare tra loro numeri con piu' di **4** bit. La Figura 6 illustra un esempio che si riferisce al confronto tra numeri binari a **8** bit; vengono utilizzati a tale proposito due comparatori a **4** bit, e cioe' **C1** che confronta il nibble basso e **C2** che confronta il nibble alto.

Figura 6

Comparazione tra numeri a 8 bit

Il confronto parte come al solito dai bit piu' significativi che in questo caso si trovano in posizione **7**; indichiamo con **Aa** e **Ba** i nibble alti, e con **Ab** e **Bb** i nibble bassi dei due numeri **A** e **B**. Se il comparatore **C2** confrontando i nibble alti verifica che **Aa** e' maggiore di **Ba** o che **Aa** e' minore di **Ba**, non ha bisogno di consultare **C1**; in questo caso **C2** ignora gli ingressi **I1**, **I2** e **I3**, e termina il confronto fornendo i risultati sulle sue uscite **Y1**, **Y2** e **Y3**.
Se invece il comparatore **C2** verifica che **Aa** e' uguale a **Ba**, allora legge i suoi ingressi **I1**, **I2**, **I3** che gli forniscono il risultato del confronto tra i nibble bassi effettuato da **C1**; come si puo' notare

dalla Figura 6, l'ingresso **I1** e' associato a **Ab** maggiore di **Bb**, l'ingresso **I2** e' associato a **Ab** uguale a **Bb** e l'ingresso **I3** e' associato a **Ab** minore di **Bb**. In base ai livelli logici assunti da **I1**, **I2** e **I3**, il comparatore **C2** fornisce il risultato finale sempre attraverso le sue uscite **Y1**, **Y2**, **Y3**; anche in questo caso si nota che la condizione **A** uguale a **B** puo' essere verificata con certezza solo dopo l'ultimo confronto che coinvolge i bit **A0** e **B0** (cioe' gli **LSB**).
Naturalmente, nell'esempio di Figura 6, gli ingressi **I1**, **I2** e **I3** del comparatore **C1** non vengono utilizzati; appare anche evidente il fatto che attraverso il collegamento in parallelo di piu' comparatori a **4** bit, si possono confrontare tra loro numeri binari di qualunque dimensione.
Un computer con architettura a **8** bit, sara' dotato di comparatori a **8** bit; un computer con architettura a **16** bit sara' dotato di comparatori a **16** bit e cosi' via. Quindi su un computer con architettura a **8** bit, non sara' possibile confrontare via hardware numeri a **16** bit; se si vuole effettuare ugualmente il confronto tra numeri a **16** bit si dovra' procedere via software scrivendo un apposito programma che naturalmente non potra' raggiungere le prestazioni del confronto via hardware. Molti linguaggi di programmazione di alto livello, mettono a disposizione delle funzioni (sottoprogrammi) che permettono di effettuare via software diverse operazioni su numeri formati da un numero di bit maggiore di quello dell'architettura del computer che si sta utilizzando.

6.3 Addizione tra numeri binari

Come abbiamo visto nei precedenti capitoli, per sommare due numeri espressi nel sistema posizionale arabo, bisogna incolonnarli in modo da far corrispondere sulla stessa colonna le cifre aventi peso identico nei due numeri stessi; sappiamo anche che la somma viene eseguita a partire dalla colonna piu' a destra che contiene le cifre meno significative dei due addendi. Questa prima somma non deve tenere conto di nessun riporto precedente; nel sommare invece le colonne successive alla prima, si dovra' tenere conto dell'eventuale riporto proveniente dalla somma della colonna precedente.
E' importante ricordare inoltre che in binario, nel sommare la prima colonna, la situazione piu' critica che si puo' presentare e':
`1 + 1`
che da' come risultato **0** con riporto di **1**; nel sommare le colonne successive alla prima, in presenza di un riporto pari a **1**, la situazione piu' critica che si puo' presentare e':
`1 + 1 + 1`
che da' come risultato **1** con riporto di **1**. In base **2** quindi e in qualsiasi altra base, il riporto massimo non puo' superare **1**.
Da quanto abbiamo appena detto, si puo' capire che la **R.C.** che dovra' eseguire la somma della prima colonna, sara' molto piu' semplice della **R.C.** che somma le colonne successive tenendo conto di eventuali riporti; per questo motivo, i circuiti che sommano le coppie di bit si suddividono in due tipi chiamati: **Half Adder** e **Full Adder**.

La Figura 7a illustra la tabella della verita' di un **Half Adder** (**H.A.**) o semisommatore binario; la Figura 7b illustra la **R.C.** che permette di implementare la somma di due bit.

Figura 7

L'**H.A.** esegue la somma relativa ai due bit che nell'addizione occupano la prima colonna (**LSB**), e produce quindi un valore a **2** bit compreso tra **00b** e **10b**; nella tabella della verita' indichiamo con **S** il bit piu' a destra, e con **C** il bit piu' a sinistra della somma appena effettuata dall'**H.A.**. Il bit **S** rappresenta una delle cifre del risultato finale (si tratta per la precisione del **LSB**); il bit **C** (**Carry**) rappresenta il riporto (**0** o **1**) di cui si dovra' tenere conto nella somma relativa alla colonna successiva dell'addizione.
Osservando la tabella della verita' dell'**H.A.** in Figura 7a, si vede subito che il contenuto della colonna **S** coincide esattamente con i livelli logici in uscita da una porta **EX-OR** in seguito alla operazione:
`A EX-OR B`
Notiamo anche che il contenuto della colonna **C** (riporto destinato alla colonna successiva), coincide esattamente con i livelli logici forniti in uscita da una porta **AND** in seguito alla operazione:
`A AND B`
In base a questi risultati, siamo in grado di definire le due funzioni che caratterizzano l'**H.A.** e che sono:
`S = A EX-OR B` e `C = A AND B`
La Figura 7b illustra la **R.C.** che soddisfa la tabella della verita' di Figura 7a; possiamo dire quindi che l'**H.A.** ha bisogno di due ingressi che rappresentano i due bit da sommare, e di due uscite che rappresentano una delle cifre del risultato finale e il riporto destinato alla colonna successiva dell'addizione.

La Figura 8a illustra la tabella della verita' di un **Full Adder** (**F.A.**) o sommatore binario; la Figura 8b illustra la **R.C.** che permette di implementare la somma di due bit piu' il riporto.

Figura 8

Input			Output	
C0	A1	B1	S1	C1
0	0	0	0	0
1	0	0	1	0
0	0	1	1	0
1	0	1	0	1
0	1	0	1	0
1	1	0	0	1
0	1	1	0	1
1	1	1	1	1

Il **F.A.** esegue la somma relativa ai due bit che nell'addizione occupano una tra le colonne successive alla prima, e deve tenere conto quindi anche del bit di riporto proveniente dalla colonna precedente; questa volta la somma produce un valore a **2** bit che sara' compreso tra **00b** e **11b**. Nella tabella della verita' indichiamo con **S** il bit piu' a destra, e con **C** il bit piu' a sinistra della somma appena effettuata dal **F.A.**; il bit **S** rappresenta una delle cifre del risultato finale, mentre il bit **C** (**Carry**) rappresenta il riporto (**0** o **1**) destinato alla colonna successiva dell'addizione.
Supponiamo di dover sommare i bit della seconda colonna di una addizione, e indichiamo con **C0** il riporto precedente, con **A1** e **B1** i due bit da sommare, con **S1** la cifra meno significativa della somma **A1+B1** e con **C1** il riporto di **A1+B1** destinato alla colonna successiva; la tabella della verita' di Figura 8a e' formata da **8** righe in quanto per ciascuno dei **4** possibili risultati della somma **A1+B1**, dobbiamo considerare i due casi **C0=0** e **C0=1** per un totale di **4x2=8** casi distinti.
Le funzioni booleane che realizzano il **F.A.** possono essere ottenute facilmente ricordando la

tecnica che si utilizza per eseguire le addizioni con carta e penna; in relazione alle colonne successive alla prima, sappiamo che prima di tutto si sommano le due cifre in colonna, dopo di che si aggiunge al risultato ottenuto il riporto precedente.
Queste due addizioni consecutive suggeriscono l'idea di utilizzare per il **F.A.**, due **H.A.** in serie come si vede in Figura 8b; il primo **H.A.** somma i due bit in colonna (**A1+B1**) producendo in uscita i risultati parziali **S'** e **C'**, mentre il secondo **H.A.** somma **S'** con il riporto precedente **C0** producendo a sua volta i risultati parziali **S''** e **C''**. Alla fine **S''** rappresenta la cifra **S1** del risultato finale, mentre i due riporti **C'** e **C''** devono fornirci il riporto **C1** destinato alla colonna successiva.
Come si verifica facilmente dal circuito di Figura 8b, i due riporti **C'** e **C''** non possono mai essere entrambi a livello logico **1** (la somma massima infatti e' **11b**), e quindi basta applicarli ai due ingressi di una porta **OR** per ottenere il riporto **C1** destinato alla colonna successiva.
Possiamo dire quindi che il **F.A.** ha bisogno di tre ingressi che rappresentano i due bit da sommare e il riporto proveniente dalla colonna precedente; le due uscite del **F.A.** forniscono invece una delle cifre del risultato finale e il riporto destinato alla colonna successiva.

Una volta realizzati l'**H.A.** e il **F.A.**, possiamo realizzare circuiti per sommare numeri binari di qualsiasi dimensione; lo schema a blocchi di Figura 9 mostra una **R.C.** per la somma tra numeri binari a **4** bit:

Figura 9

La **R.C.** di Figura 9 utilizza un **H.A.** e tre **F.A.**; l'**H.A.** somma i bit della prima colonna (**LSB**), mentre i tre **F.A.** sommano i bit delle tre colonne successive. Il riporto finale **C3** contiene il valore (**0** o **1**) che verra' assegnato al **Carry Flag**; come gia' sappiamo, grazie a **CF** possiamo sommare via software numeri binari di qualsiasi ampiezza in bit.
Anche nel caso dell'addizione (e di tutte le altre operazioni), bisogna ricordare che un computer con architettura ad esempio a **16** bit, sara' dotato di circuiti in grado di sommare via hardware numeri a **16** bit; se vogliamo sommare tra loro numeri a **32** bit, dobbiamo procedere via software suddividendo i bit degli addendi in gruppi da **16**.
Nei precedenti capitoli abbiamo anche appurato che grazie all'aritmetica modulare, una **R.C.** come quella di Figura 9 puo' sommare indifferentemente numeri con o senza segno; alla fine dell'addizione, se stiamo operando sui numeri senza segno dobbiamo consultare **CF** per conoscere il riporto finale, mentre se stiamo operando sui numeri con segno dobbiamo consultare **OF** e **SF** per sapere se il risultato e' valido e per individuarne il segno. Come si puo' facilmente verificare attraverso la Figura 9, si ha:
`CF = C3`, `SF = S3` e `OF = (A3 EX-NOR B3) AND (A3 EX-OR B3 EX-OR S3)`

6.4 Sottrazione tra numeri binari

Per la sottrazione valgono considerazioni analoghe a quelle svolte per l'addizione; sottraendo i bit della colonna piu' a destra non si deve tenere conto di eventuali prestiti chiesti dalle colonne precedenti; sottraendo invece i bit delle colonne successive alla prima, bisogna tenere conto degli eventuali prestiti chiesti dalle colonne precedenti. Nel primo caso, la situazione piu' critica che puo' presentarsi e':

```
0 - 1
```

che da' come risultato **1** con prestito di **1** dalle colonne successive; nel secondo caso, in presenza di un prestito pari a **1** richiesto dalla colonna precedente, la situazione piu' critica che puo' presentarsi e':

```
0 - 1 - 1
```

che da' come risultato **0** con prestito di **1** dalle colonne successive.
In qualsiasi base numerica quindi, il prestito massimo non puo' superare **1**; per gestire questi due casi possibili vengono impiegati due circuiti chiamati **Half Subtractor** e **Full Subtractor**.

La Figura 10a illustra la tabella della verita' di un **Half Subtractor** (**H.S.**) o semisottrattore binario; la Figura 10b illustra la **R.C.** che permette di implementare la sottrazione tra due bit.

Figura 10

Input		Output	
A	B	D	P
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

L'**H.S.** calcola la differenza tra i due bit che nella sottrazione occupano la prima colonna (**LSB**), e produce quindi come risultato un valore a **1** bit compreso tra **0b** e **1b**; nella tabella della verita' indichiamo questo bit con **D**. Il bit **D** rappresenta una delle cifre del risultato finale (in questo caso si tratta del **LSB**); indichiamo poi con **P** il bit che rappresenta il prestito (**0** o **1**) richiesto alle colonne successive.
Osservando la tabella della verita' dell'**H.S.** in Figura 10a, si vede subito che il contenuto della colonna **D** coincide esattamente con i livelli logici in uscita da una porta **EX-OR** in seguito alla operazione:

```
A EX-OR B
```

Notiamo anche che il contenuto della colonna **P** (prestito richiesto alle colonne successive), coincide esattamente con i livelli logici forniti in uscita da una porta **AND** in seguito alla operazione:

```
NOT(A) AND B
```

In base a questi risultati, siamo in grado di definire le due funzioni che caratterizzano l'**H.S.** e che sono:

`D = A EX-OR B` e `P = NOT(A) AND B`

La Figura 10b illustra la **R.C.** che soddisfa la tabella della verita' di Figura 10a; possiamo dire quindi che l'**H.S.** ha bisogno di due ingressi che rappresentano i due bit da sottrarre, e di due uscite che rappresentano una delle cifre del risultato finale (in questo caso si tratta dell'**LSB**) e il prestito richiesto alle colonne successive della sottrazione.
La Figura 11a illustra la tabella della verita' di un **Full Subtractor** (**F.S.**) o sottrattore binario; la

Figura 11b illustra la **R.C.** che permette di implementare la differenza tra due bit tenendo conto della presenza di un eventuale prestito.

Input			Output	
P0	A1	B1	D1	P1
0	0	0	0	0
1	0	0	1	1
0	0	1	1	1
1	0	1	0	1
0	1	0	1	0
1	1	0	0	0
0	1	1	0	0
1	1	1	1	1

Il **F.S.** calcola la differenza tra i due bit che nella sottrazione occupano una tra le colonne successive alla prima, e produce quindi come risultato un valore a **1** bit compreso tra **0b** e **1b**; nella tabella della verita' indichiamo questo bit con **D**. Il bit **D** rappresenta una delle cifre del risultato finale; indichiamo poi con **P** il bit che rappresenta il prestito (**0** o **1**) richiesto alle colonne successive.
Supponiamo di dover calcolare la differenza tra i bit della seconda colonna di una sottrazione, e indichiamo con **P0** il prestito precedente, con **A1** e **B1** i due bit da sottrarre, con **D1** la cifra meno significativa della sottrazione **A1-B1** e con **P1** il prestito richiesto da **A1-B1** alle colonne successive; la tabella della verita' di Figura 11a e' formata da **8** righe in quanto per ciascuno dei **4** possibili risultati della differenza **A1-B1**, dobbiamo considerare i due casi **P0=0** e **P0=1** per un totale di **4x2=8** casi distinti.
Anche in questo caso le funzioni booleane che realizzano il **F.S.** possono essere ottenute facilmente ricordando la tecnica che si utilizza per eseguire le sottrazioni con carta e penna; in relazione alle colonne successive alla prima, possiamo iniziare dalla sottrazione tra le due cifre in colonna, dopo di che sottraiamo al risultato ottenuto il prestito precedente.
Queste due sottrazioni consecutive suggeriscono l'idea di utilizzare per il **F.S.**, due **H.S.** in serie come si vede in Figura 11b; il primo **H.S.** sottrae i due bit in colonna (**A1-B1**) producendo in uscita i risultati parziali **D'** e **P'**, mentre il secondo **H.S.** sottrae **D'** con il prestito precedente **P0** producendo a sua volta i risultati parziali **D''** e **P''**. Alla fine **P''** rappresenta la cifra **D1** del risultato finale, mentre i due prestiti **P'** e **P''** devono fornirci il riporto **P1** richiesto alle colonne successive.
Come si verifica facilmente dal circuito di Figura 11b, i due prestiti **P'** e **P''** non possono mai essere entrambi a livello logico **1** (il prestito massimo infatti e' **1**), e quindi basta applicarli ai due ingressi di una porta **OR** per ottenere il prestito **P1** richiesto alle colonne successive.
Possiamo dire quindi che il **F.S.** ha bisogno di tre ingressi che rappresentano i due bit da sottrarre e il prestito richiesto dalla colonna precedente; le due uscite del **F.S.** forniscono invece una delle cifre del risultato finale e il prestito richiesto alle colonne successive.

Una volta realizzati l'**H.S.** e il **F.S.**, possiamo realizzare circuiti per sottrarre numeri binari di qualsiasi dimensione; lo schema a blocchi di Figura 12 mostra una **R.C.** per la sottrazione tra numeri binari a **4** bit:

Figura 12

La **R.C.** di Figura 12 utilizza un **H.S.** e tre **F.S.**; l'**H.S.** sottrae i bit della prima colonna (**LSB**), mentre i tre **F.S.** sottraggono i bit delle tre colonne successive. Il prestito finale **P3** contiene il valore (**0** o **1**) che verra' assegnato al **Carry Flag**; come gia' sappiamo, grazie a **CF** possiamo sottrarre via software numeri binari di qualsiasi ampiezza in bit.
Un computer con architettura ad esempio a **16** bit, sara' dotato di circuiti in grado di sottrarre via hardware numeri a **16** bit; se vogliamo sottrarre tra loro numeri a **32** bit, dobbiamo procedere via software suddividendo i bit del minuendo e del sottraendo in gruppi da **16**.
Nei precedenti capitoli abbiamo anche appurato che grazie all'aritmetica modulare, una **R.C.** come quella di Figura 12 puo' sottrarre indifferentemente numeri con o senza segno; alla fine della sottrazione, se stiamo operando sui numeri senza segno dobbiamo consultare **CF** per conoscere il prestito finale, mentre se stiamo operando sui numeri con segno dobbiamo consultare **OF** e **SF** per sapere se il risultato e' valido e per individuarne il segno. Come si puo' facilmente verificare attraverso la Figura 12, si ha:
`CF = P3`, `SF = D3` e `OF = (A3 EX-NOR B3) AND (A3 EX-OR B3 EX-OR D3)`

6.5 Circuiti complementatori

Nella precedente sezione sono stati descritti i circuiti che eseguono la sottrazione tra numeri binari; nella realizzazione pratica di questi circuiti puo' essere seguita anche un'altra strada. Osserviamo infatti che, dati i due numeri binari **n1** e **n2**, possiamo scrivere:
`n1 - n2 = n1 + (-n2)`
In sostanza, la differenza tra **n1** e **n2** e' pari alla somma tra **n1** e l'opposto di **n2**; come gia' sappiamo, l'opposto di un numero binario si ottiene effettuando il complemento a **2** del numero stesso. Il complemento a **2** (**NEG**) di un numero binario **n** si ottiene invertendo tutti i suoi bit (**NOT**) e sommando **1** al risultato ottenuto; possiamo scrivere quindi:
`NEG(n) = NOT(n) + 1`
Per realizzare nel modo piu' semplice il complemento a **1** di un numero, possiamo utilizzare delle semplici porte **NOT** che come sappiamo producono in uscita un livello logico che e' l'opposto di quello in ingresso; se entra un segnale alto, esce un segnale basso, mentre se entra un segnale basso, esce un segnale alto. Quindi se vogliamo complementare ad esempio numeri binari a **16** bit, possiamo utilizzare **16** porte **NOT**; ciascuna di queste porte inverte uno dei bit del numero da complementare, e alla fine otteniamo in uscita il numero in ingresso con tutti i bit invertiti.
Il circuito complementatore appena descritto, e' molto semplice; nella pratica pero' si segue un'altra strada legata come al solito alla semplificazione circuitale. Invece di realizzare una **R.C.** per ogni funzione booleana, si preferisce realizzare delle **R.C.** leggermente piu' complesse, che permettono pero' di svolgere diverse funzioni con lo stesso circuito; in Figura 13 ad esempio, vediamo un

dispositivo a **4** bit che svolge **4** funzioni differenti a seconda dei livelli logici assunti dai cosiddetti **ingressi di selezione** che sono **S0** e **S1**:

Figura 13

Select		Input				Output			
S0	S1	I0	I1	I2	I3	Y0	Y1	Y2	Y3
0	0	A0	A1	A2	A3	$\overline{A0}$	$\overline{A1}$	$\overline{A2}$	$\overline{A3}$
0	1	A0	A1	A2	A3	A0	A1	A2	A3
1	0	--	--	--	--	1	1	1	1
1	1	--	--	--	--	0	0	0	0

Dalla tabella della verita' di Figura 13 si vede che attribuendo ai due ingressi **S0** e **S1** i **4** (**2^2**) possibili valori binari **00b**, **01b**, **10b** e **11b**, si possono ottenere in uscita **4** differenti elaborazioni dello stesso nibble in ingresso. In particolare:

***** Per **S0 = 0** e **S1 = 0** si ottiene in uscita il complemento a **1** del nibble in ingresso.
***** Per **S0 = 0** e **S1 = 1** si ottiene in uscita lo stesso nibble in ingresso.
***** Per **S0 = 1** e **S1 = 0** si ottiene in uscita il nibble **1111b**.
***** Per **S0 = 1** e **S1 = 1** si ottiene in uscita il nibble **0000b**.

In questi ultimi due casi, il nibble in ingresso e' ininfluente; con lo stesso circuito quindi realizziamo **4** funzioni differenti che altrimenti avrebbero richiesto **4** circuiti diversi.
I circuiti complementatori sono molto importanti in quanto vengono largamente utilizzati dalla **CPU** in molte operazioni; ricordiamo ad esempio che la **CPU** esegue moltiplicazioni e divisioni solo sui numeri senza segno. Eventuali numeri negativi vengono prima di tutto convertiti nei corrispondenti numeri positivi; per poter effettuare queste conversioni vengono appunto utilizzati i circuiti complementatori.

6.6 Moltiplicazione tra numeri binari

Nei precedenti capitoli e' stato detto che la **CPU** effettua la moltiplicazione attraverso un metodo fortemente basato sullo stesso procedimento che utilizziamo noi esseri umani quando eseguiamo questa operazione con carta e penna; abbiamo anche visto che la moltiplicazione provoca un cambiamento di modulo, per cui la **CPU** ha bisogno di sapere se vogliamo operare nell'insieme dei numeri senza segno o nell'insieme dei numeri con segno.
Moltiplicando tra loro due numeri binari a **n** bit, otteniamo un risultato interamente rappresentabile attraverso un numero binario a **2n** bit; partiamo allora dai numeri interi senza segno a **4** bit (nibble), e supponiamo di voler calcolare ad esempio:

```
10 x 13 = 130
```

Traducendo tutto in binario, e applicando lo stesso procedimento che si segue con carta e penna, otteniamo la situazione mostrata in Figura 14:

Figura 14

```
1010 x 1101 =
-------------
      1010
     0000
    1010
   1010
-------------
 10000010
```

Osserviamo che moltiplicando tra loro due numeri a **4** bit, otteniamo un risultato che richiede al massimo **8** bit; la **CPU** tiene conto di questo aspetto, e quindi e' impossibile che la moltiplicazione possa provocare un overflow.
Notiamo anche l'estrema semplicita' del calcolo dei prodotti parziali; considerando il primo fattore **1010b**, si possono presentare solamente i due casi seguenti:
a) 1010b x 0b = 0000b
b) 1010b x 1b = 1010b

La Figura 15 mostra in forma simbolica la moltiplicazione tra i due nibble **A3A2A1A0** e **B3B2B1B0**:

Figura 15

```
         (A3 A2 A1 A0) x (B3 B2 B1 B0) =
----------------------------------------------
                        A3xB0 A2xB0 A1xB0 A0xB0
                  A3xB1 A2xB1 A1xB1 A0xB1
            A3xB2 A2xB2 A1xB2 A0xB2
      A3xB3 A2xB3 A1xB3 A0xB3
----------------------------------------------
  P7    P6    P5    P4    P3    P2    P1    P0
```

Cominciamo con l'osservare che le singole cifre dei prodotti parziali non sono altro che moltiplicazioni tra singoli bit; nel precedente capitolo abbiamo visto che il prodotto binario e' ottenibile attraverso una porta **AND** a **2** ingressi. Nel caso di Figura 15 abbiamo ad esempio:

```
A3 x B2 = A3 AND B2
```

Osserviamo ora che la cifra **P0** del prodotto finale (cifra meno significativa) si ottiene direttamente da **A0xB0**; abbiamo quindi:

```
P0 = A0 AND B0
```

La cifra **P1** del prodotto finale e' data da:

```
P1 = (A0 AND B1) + (A1 AND B0)
```

Per svolgere questa somma utilizziamo un **H.A.** che produce in uscita la cifra **P1** e il riporto **C1** destinato alla colonna successiva.
La cifra **P2** del prodotto finale e' data da:

```
P2 = (A0 AND B2) + (A1 AND B1) + (A2 AND B0) + C1
```

Per svolgere la prima somma **(A0 AND B2) + (A1 AND B1)** utilizziamo un **H.A.** che produce in uscita la somma parziale **P2'** e il riporto **C2'** destinato alla colonna successiva; per svolgere la seconda somma tra **P2'** e **(A2 AND B0)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C1** derivante dal calcolo di **P1**, e produce in uscita la cifra **P2** e il riporto **C2''** destinato alla colonna successiva.
La cifra **P3** del prodotto finale e' data da:

```
P3 = (A0 AND B3) + (A1 AND B2) + (A2 AND B1) + (A3 AND B0) + C2' + C2''
```

Per svolgere la prima somma **(A0 AND B3) + (A1 AND B2)** utilizziamo un **H.A.** che produce in uscita la somma parziale **P3'** e il riporto **C3'** destinato alla colonna successiva; per svolgere la seconda somma tra **P3'** e **(A2 AND B1)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C2'** derivante dal calcolo di **P2**, e produce in uscita la somma parziale **P3''** e il riporto **C3''** destinato alla colonna successiva. Per svolgere la terza somma tra **P3''** e **(A3 AND B0)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C2''** derivante dal calcolo di **P2**, e produce in uscita la cifra **P3** e il riporto **C3'''** destinato alla colonna successiva.
La cifra **P4** del prodotto finale e' data da:

```
P4 = (A1 AND B3) + (A2 AND B2) + (A3 AND B1) + C3' + C3'' + C3'''
```

Per svolgere la prima somma **(A1 AND B3) + (A2 AND B2)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C3'** derivante dal calcolo di **P3**, e produce in uscita la somma parziale **P4'** e il riporto **C4'** destinato alla colonna successiva; per svolgere la seconda somma tra **P4'** e **(A3 AND B1)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C3''** derivante dal calcolo di **P3**, e produce in uscita la somma parziale **P4''** e il riporto **C4''** destinato alla colonna successiva. A questo punto utilizziamo un **H.A.** per sommare **P4''** con il riporto **C3'''** derivante dal calcolo di **P3**; in questo modo otteniamo in uscita dall'**H.A.** la cifra **P4** e il riporto **C4'''** destinato alla colonna successiva.
La cifra **P5** del prodotto finale e' data da:

```
P5 = (A2 AND B3) + (A3 AND B2) + C4' + C4'' + C4'''
```

Per svolgere questa somma utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C4'** derivante dal calcolo di **P4**, e produce in uscita la somma parziale **P5'** e il riporto **C5'** destinato alla colonna successiva. A questo punto utilizziamo un **F.A.** per sommare **P5'** con i riporti **C4''** e **C4'''** derivanti dal calcolo di **P4**; in questo modo otteniamo in uscita dal **F.A.** la cifra **P5** e il riporto **C5''** destinato alla colonna successiva.
La cifra **P6** del prodotto finale e' data da:

```
P6 = (A3 AND B3) + C5' + C5''
```

Utilizziamo allora un **F.A.** che riceve in ingresso anche i riporti **C5'** e **C5''** derivanti dal calcolo di **P5**; in questo modo otteniamo in uscita dal **F.A.** la cifra **P6** e il riporto **C6** destinato alla colonna successiva.
Naturalmente il riporto **C6** non e' altro che la cifra **P7** del prodotto finale; quest'ultimo passaggio conferma che nella moltiplicazione l'overflow e' impossibile.

Applicando alla lettera i concetti appena esposti, otteniamo la **R.C.** di Figura 16 che permette di moltiplicare numeri interi senza segno a **4** bit:

Figura 16

La **R.C.** appena descritta, pur essendo perfettamente funzionante, ha un valore puramente teorico; nella pratica infatti si utilizzano **R.C.** che pur essendo basate sul circuito di Figura 16, presentano notevoli ottimizzazioni che hanno lo scopo di aumentare la velocita' di calcolo. In ogni caso appare chiaro il fatto che queste ottimizzazioni, per quanto spinte possano essere, danno luogo ad **R.C.** nettamente piu' complesse di quelle utilizzate dalla **CPU** per le addizioni e per le sottrazioni; proprio per questo motivo, la moltiplicazione impone alla **CPU** un tempo di calcolo nettamente superiore a quello necessario per l'esecuzione delle addizioni e delle sottrazioni.

La **R.C.** di Figura 16 puo' essere utilizzata anche per la moltiplicazione tra numeri interi con segno; a tale proposito bisogna prima di tutto cambiare di segno gli eventuali fattori negativi. In teoria questo metodo funziona, ma la **R.C.** che ne deriva tende a diventare molto piu' complessa di quella di Figura 16; proprio per questo motivo, molte **CPU** utilizzano **R.C.** appositamente concepite per operare direttamente sui numeri interi con segno.

6.7 Divisione tra numeri binari

Anche la divisione viene effettuata dalla **CPU** attraverso un metodo basato sullo stesso procedimento che utilizziamo noi esseri umani quando eseguiamo questa operazione con carta e penna; come viene mostrato in seguito, l'algoritmo utilizzato dalla **CPU** comporta un cambiamento di modulo, per cui e' necessario specificare se vogliamo operare nell'insieme dei numeri senza segno o nell'insieme dei numeri con segno.
Dividendo tra loro due numeri binari a **n** bit, otteniamo un quoziente minore o uguale al dividendo, e quindi interamente rappresentabile attraverso un numero binario a **n** bit; il resto inoltre e' minore o uguale al divisore, per cui anch'esso e' interamente rappresentabile attraverso un numero binario a **n** bit.

Partiamo come al solito dai numeri interi senza segno a **4** bit (nibble), e supponiamo di voler calcolare ad esempio:

```
14 / 5 = Q = 2, R = 4
```

Traducendo tutto in binario, e applicando lo stesso procedimento che si segue con carta e penna, otteniamo la situazione mostrata in Figura 17:

Figura 17

```
1110 / 101 =
101    ------
----   10
 100
 000
----
 100
```

Analizzando la Figura 17 possiamo riassumere i passi necessari per il classico algoritmo di calcolo della divisione:

1) A partire da sinistra si seleziona nel dividendo un numero di bit pari al numero di bit del divisore; la sequenza di bit cosi' ottenuta rappresenta il resto parziale della divisione.
2) Se e' possibile sottrarre il divisore dal resto parziale senza chiedere un prestito, si aggiunge un bit di valore **1** alla destra del quoziente parziale; il risultato della sottrazione rappresenta il nuovo resto parziale. Se invece non e' possibile sottrarre il divisore dal resto parziale senza chiedere un prestito,

si aggiunge un bit di valore **0** alla destra del quoziente parziale.
3) Se e' possibile, si aggiunge alla destra del resto parziale un nuovo bit del dividendo e si ricomincia dal punto **2**; se invece non ci sono altri bit del dividendo da aggiungere al resto parziale, la divisione e' terminata con quoziente finale pari all'ultimo quoziente parziale, e resto finale pari all'ultimo resto parziale.

Per poter realizzare un circuito logico capace di effettuare l'operazione di divisione, dobbiamo elaborare il precedente algoritmo in modo da renderlo generale e ripetitivo; dobbiamo cioe' fare in modo che la **CPU** calcoli la divisione ripetendo (iterando) uno stesso procedimento per un determinato numero di volte. Per raggiungere questo obiettivo si utilizza un algoritmo perfettamente equivalente a quello appena illustrato; in riferimento come al solito ai numeri interi senza segno a **4** bit, i passi da compiere sono i seguenti:

1) Si aggiungono quattro **0** alla sinistra del dividendo che viene cosi' convertito in un numero a **8** bit; il numero cosi' ottenuto rappresenta il resto parziale della divisione.
2) Si aggiungono quattro **0** alla destra del divisore che viene cosi' convertito in un numero a **8** bit; le cifre del divisore in sostanza vengono shiftate a sinistra di quattro posti.
3) Le cifre del divisore vengono shiftate a destra di una posizione.
4) Si prova a sottrarre il divisore dal resto parziale senza chiedere un prestito, e se cio' e' possibile si aggiunge un bit di valore **1** alla destra del quoziente parziale, mentre il risultato della sottrazione rappresenta il nuovo resto parziale; se invece la sottrazione non e' possibile, si aggiunge un bit di valore **0** alla destra del quoziente parziale.
5) Si salta al punto **3** e si ripete questo procedimento per **4** volte, pari cioe' al numero di bit dei due operandi; l'ultimo quoziente parziale ottenuto rappresenta il quoziente finale della divisione, mentre l'ultimo resto parziale ottenuto rappresenta il resto finale della divisione.

Proviamo ad applicare questo algoritmo alla precedente divisione di **1110b** per **0101b**; il risultato che si ottiene viene mostrato dalla Figura 18:

Figura 18

```
Il dividendo 1110b diventa 00001110b
Il divisore  0101b diventa 01010000b

1a) il divisore diventa 00101000b
1b) 00001110b - 00101000b non e' possibile e quindi Q' = 0b

2a) il divisore diventa 00010100b
2b) 00001110b - 000101000b non e' possibile e quindi Q' = 00b

3a) il divisore diventa 00001010b
3b) 00001110b - 00001010b = 00000100b e quindi Q' = 001b

4a) il divisore diventa 00000101b
4b) 00000100b - 00000101b non e' possibile e quindi Q' = 0010b

Quoziente finale Q = 0010b
Resto finale     R = 0100b
```

In riferimento quindi ai numeri interi senza segno a **4** bit, possiamo dire che l'algoritmo appena illustrato consiste nel ripetere per **4** volte uno stesso procedimento che comprende uno shift di un posto verso destra delle cifre del divisore, e un tentativo di eseguire una sottrazione tra il resto parziale e lo stesso divisore.
Nel caso della moltiplicazione, abbiamo risolto il problema attraverso una **R.C.** relativamente semplice; nel caso della divisione invece la situazione si presenta piu' complessa. La necessita' di

dover stabilire se effettuare o meno la sottrazione, rende necessario un apposito dispositivo capace di prendere una tale decisione; ci serve cioe' un dispositivo derivato dal **F.S.** e dotato di una linea di controllo attraverso la quale si possa abilitare o meno la sottrazione. Questo dispositivo prende il nome di **Conditional Subtractor** (**C.S.**) o sottrattore condizionale; lo schema circuitale del **C.S.** viene mostrato in Figura 19a, mentre lo schema simbolico viene mostrato in Figura 19b:

Come si puo' notare, la struttura del **C.S.** e' del tutto simile alla struttura del **F.S.**; sono presenti infatti l'ingresso **Ai** per il bit del minuendo, l'ingresso **Bi** per il bit del sottraendo, l'ingresso **Cin** (Carry In) per il prestito richiesto dalla colonna precedente, l'uscita **Cout** (Carry Out) per il prestito richiesto alla colonna successiva e l'uscita **D** per il risultato finale che nel caso del **F.S.** e' dato da:

```
D = Ai - (Bi + Cin)
```

In Figura 19 notiamo pero' anche la presenza della linea **Enable** che attraversa il **C.S.** da **Ein** a **Eout**; proprio attraverso questa linea e' possibile modificare il comportamento del **C.S.**. Come si puo' vedere in Figura 19a, l'ingresso **Ai** e l'uscita **D'** del **F.S.** rappresentano i **2** ingressi di un **MUX** da **2** a **1**; analizzando allora questo circuito possiamo dire che:

a) Se la linea **Enable** e' a livello logico **0**, viene abilitato l'ingresso **D'** del **MUX**, per cui il **C.S.** calcola:

```
D = Ai - (Bi + Cin)
```

b) Se la linea **Enable** e' a livello logico **1**, viene abilitato l'ingresso **Ai** del **MUX**, per cui il **C.S.** calcola:

```
D = Ai
```

Possiamo dire quindi che se la linea **Enable** e' a livello logico **0**, il **C.S.** effettua normalmente la sottrazione; se invece la linea **Enable** e' a livello logico **1**, si ottiene sull'uscita **D** il bit **Ai** del minuendo. Si tenga presente che in Figura 19 la linea **Enable** non ha niente a che vedere con il segnale di abilitazione **E** del **MUX** di Figura 1.
A questo punto si presenta il problema di come pilotare la linea **Enable**; in sostanza, la linea **Enable** deve essere portata a livello logico **0** solo quando il minuendo e' maggiore o uguale al sottraendo. Il metodo che viene utilizzato, consiste nello sfruttare l'uscita **Cout** che viene collegata all'ingresso **Ein**; per capire il funzionamento di questo sistema, analizziamo in dettaglio il funzionamento del **C.S.** di Figura 19:

***** Il **C.S.** effettua la normale sottrazione **D=Ai-(Bi+Cin)**, e ottiene come sappiamo **Cout=0** se non c'e' nessun prestito (**Ai** maggiore o uguale a **Bi**); se invece si verifica un prestito (**Ai** minore di **Bi**), il **C.S.** ottiene **Cout=1**.

***** A questo punto, come e' stato detto in precedenza, l'uscita **Cout** viene collegata a **Ein**, per cui **Ein=Cout=0** conferma il risultato della sottrazione, mentre **Ein=Cout=1** lo annulla e pone **D=Ai**.

In base alle considerazioni appena esposte, si intuisce facilmente che per ottenere la sottrazione condizionale tra due numeri binari a **4** bit, basta collegare in serie **4 C.S.**; il circuito che si ottiene viene mostrato in Figura 20:

Figura 20

Supponiamo ad esempio di avere **A=1110b** e **B=0101b**; in questo caso si vede subito che:

***** Il primo **C.S.** (quello piu' a destra) pone **Cout=1**.
***** Il secondo **C.S.** pone **Cout=0**.
***** Il terzo **C.S.** pone **Cout=0**.
***** Il quarto **C.S.** pone **Cout=0**.

La linea **Enable** si porta quindi a livello logico **0**, e il risultato della sottrazione (**1001b**) viene confermato; se il quarto **C.S.** avesse prodotto **Cout=1**, la linea **Enable** si sarebbe portata a livello logico **1** annullando il risultato della sottrazione e producendo quindi in uscita il nibble **A=1110b**.
In quest'ultimo caso e' come se il minuendo venisse ripristinato (**restored**); proprio per questo motivo, l'algoritmo di calcolo della divisione con i **C.S.** prende il nome di **restoring division** (divisione con ripristino).
Applicando le considerazioni appena esposte, possiamo implementare finalmente il circuito per la divisione tra numeri interi senza segno a **4** bit; ricordando che l'algoritmo esposto in Figura 18 prevede un procedimento ripetitivo costituito da uno shift di un posto verso destra delle cifre del divisore e da un tentativo di sottrazione tra resto parziale e divisore stesso, si ottiene il circuito mostrato in Figura 21:

Figura 21

Gli ingressi **a0**, **a1**, **a2**, **a3**, **a4**, **a5** e **a6** contengono i **4** bit del dividendo e rappresentano anche il resto parziale; naturalmente gli ingressi **a4**, **a5** e **a6** vengono portati a livello logico basso. Gli ingressi **b0**, **b1**, **b2** e **b3** contengono i **4** bit del divisore; come si puo' notare, il divisore si trova shiftato a sinistra di **3** posizioni anziche' di **4** in quanto (Figura 18) **4** shift a sinistra e uno a destra equivalgono a **3** shitf a sinistra.
La riga piu' in alto dei **C.S.** esegue la sottrazione tra **a6a5a4a3** e **b3b2b1b0**; se alla fine non si verifica nessun prestito il risultato della sottrazione viene confermato, mentre in caso contrario viene ripristinato il minuendo, e il risultato prodotto in uscita e' **a6a5a4a3**. Si nota subito che l'uscita **q3** vale **1** se il minuendo e' maggiore o uguale al sottraendo, e **0** in caso contrario; possiamo dire quindi che **q3** non e' altro che uno dei **4** bit del quoziente finale.
Le considerazioni appena esposte si applicano naturalmente anche alle altre tre righe dei **C.S.**; terminata la divisione, sulle quattro uscite **q0**, **q1**, **q2** e **q3** preleviamo i **4** bit del quoziente finale, mentre sulle quattro uscite **r0**, **r1**, **r2** e **r3** preleviamo i **4** bit del resto finale.

Come al solito, il circuito di Figura 21 pur essendo perfettamente funzionante ha un valore puramente teorico; nella pratica infatti si utilizza una versione elaborata e notevolmente ottimizzata del circuito appena descritto. Come e' facile intuire pero', la divisione si presenta come una operazione particolarmente complessa per la **CPU**; infatti la divisione comporta per la **CPU** un tempo di calcolo superiore persino a quello necessario per la moltiplicazione (soprattutto nel caso di restoring delle varie sottrazioni).

Il circuito di Figura 21 puo' essere utilizzato anche per la divisione tra numeri interi con segno; a tale proposito bisogna prima di tutto cambiare di segno gli eventuali operandi negativi. In relazione alla divisione la **CPU** ha bisogno di distinguere tra numeri con o senza segno in quanto abbiamo visto che per poter utilizzare il circuito di Figura 21, e' necessario modificare l'ampiezza in bit del dividendo e del divisore; tutto cio' dimostra che il metodo di calcolo della divisione utilizzato dalla **CPU**, provoca un cambiamento di modulo.

6.8 Scorrimento dei bit di un numero binario

Le considerazioni esposte in precedenza ci hanno permesso di constatare che la moltiplicazione e la divisione sono operazioni particolarmente impegnative per la **CPU**, e richiedono tempi di calcolo notevolmente superiori a quelli necessari per l'addizione o per la sottrazione; purtroppo nel caso generale non e' possibile ovviare a questo problema. Nei precedenti capitoli pero' abbiamo visto che per la moltiplicazione e la divisione esiste un caso particolare che ci permette di velocizzare notevolmente queste operazioni; esaminiamo allora questo caso particolare che e' legato ancora una volta alla struttura dei sistemi di numerazione posizionali.

6.8.1 Moltiplicazione di un numero intero binario per una potenza intera del 2

Nel caso ad esempio della base **10**, abbiamo visto che moltiplicare un numero intero **n** per **10** (cioe' per la base), equivale ad aggiungere uno zero alla destra del numero stesso. Tutte le cifre di **n** scorrono verso sinistra di una posizione aumentando quindi il loro peso; infatti la moltiplicazione di **n** per **10** trasforma le unita' di **n** in decine, le decine in centinaia, le centinaia in migliaia e cosi' via. Analogamente, moltiplicare **n** per **100** (10^2) equivale ad aggiungere due zeri alla destra di **n** facendo scorrere tutte le sue cifre di due posizioni verso sinistra; moltiplicare in generale **n** per 10^m equivale ad aggiungere **m** zeri alla destra di **n** facendo scorrere tutte le sue cifre di **m** posizioni verso sinistra. Nel caso ancora piu' generale di un numero intero **n** espresso in base **b** qualunque, moltiplicare **n** per b^m equivale ad aggiungere **m** zeri alla destra di **n**, facendo scorrere tutte le sue cifre di **m**

posizioni verso sinistra. Possiamo dire allora che ogni volta che uno dei due fattori puo' essere espresso sotto forma di potenza intera della base, la moltiplicazione si traduce in uno scorrimento verso sinistra delle cifre dell'altro fattore, di un numero di posizioni pari all'esponente della base; risulta evidente che in questo caso, i calcoli diventano estremamente semplici e veloci rispetto al metodo tradizionale.
Proprio per questo motivo, abbiamo visto che le **CPU** della famiglia **80x86** forniscono l'istruzione **SHL** (**Shift Logical Left**) per gli scorrimenti verso sinistra dei bit di un numero intero senza segno, e l'istruzione **SAL** (**Shift Arithmetic Left**) per gli scorrimenti verso sinistra dei bit di un numero intero con segno; siccome pero' il segno viene codificato nel **MSB** di un numero binario, le due istruzioni **SHL** e **SAL** sono assolutamente equivalenti (e quindi interscambiabili).

Per chiarire questi concetti molto importanti, e' necessario tenere presente che le considerazioni appena esposte hanno un valore puramente teorico; in teoria infatti possiamo prendere un numero **n** e possiamo far scorrere i suoi bit di migliaia di posizioni verso sinistra o verso destra. Nel caso del computer pero' il discorso cambia radicalmente; non bisogna dimenticare infatti che la **CPU** gestisce i numeri binari assegnando a ciascuno di essi una ampiezza fissa in bit. Ogni numero binario inoltre viene sistemato in una locazione ad esso riservata; consideriamo ad esempio la situazione mostrata in Figura 22:

Figura 22
0 0 1 0 0 1 1 1

Questa figura mostra una locazione da **8** bit riservata al numero binario **n=00100111b**; il numero binario **n** si trova confinato all'interno della sua locazione da **8** bit, e tutte le modifiche compiute su di esso, devono produrre un risultato interamente rappresentabile attraverso questi **8** bit.
Analizzando allora la Figura 22 si intuisce subito che eccedendo nello scorrimento verso sinistra dei bit di **n**, si rischia di ottenere un risultato sbagliato; il problema che si verifica e' dovuto chiaramente al fatto che un numero eccessivo di scorrimenti puo' provocare la fuoriuscita (dalla parte sinistra della locazione di Figura 22) di cifre significative per il risultato finale.
Supponiamo ad esempio di far scorrere (con **SHL** o con **SAL**) ripetutamente di un posto verso sinistra i bit del numero binario di Figura 22; in questo modo otteniamo la seguente successione:
`01001110b, 10011100b, 00111000b, 01110000b, ...`
Tutti gli scorrimenti si applicano agli **8** bit della locazione di Figura 22, e devono quindi produrre un risultato interamente contenibile nella locazione stessa; e' chiaro quindi che dopo un certo numero di scorrimenti, comincera' a verificarsi il trabocco da sinistra di cifre significative del risultato.
Nell'insieme dei numeri senza segno il numero binario **00100111b** equivale a **39**; il primo scorrimento produce il risultato **78** che e' corretto in quanto rappresenta il prodotto della moltiplicazione di **39** per **2**. Il secondo scorrimento produce il risultato **156** che e' corretto in quanto rappresenta il prodotto della moltiplicazione di **39** per $\mathbf{2^2}$; il terzo scorrimento produce il risultato **56** che e' chiaramente sbagliato. L'errore e' dovuto al fatto che moltiplicando **39** per $\mathbf{2^3}$ si ottiene un numero a **9** bit che non e' piu' contenibile nella locazione di Figura 22; il bit piu' significativo del risultato (di valore **1**) trabocca da sinistra e invalida il risultato finale.

Anche nell'insieme dei numeri con segno il numero binario **00100111b** equivale a **+39**; il primo scorrimento produce il risultato **+78** che e' corretto in quanto rappresenta il prodotto della moltiplicazione di **+39** per **2**. Il secondo scorrimento produce il risultato **-100** che e' chiaramente sbagliato; l'errore questa volta e' dovuto al fatto che moltiplicando **+39** per $\mathbf{2^2}$ si ottiene un numero positivo (**+156**) piu' grande di **+127** che e' il massimo numero positivo rappresentabile con **8** bit.
Osserviamo infatti che il bit di segno che inizialmente valeva **0**, e' traboccato da sinistra, e il suo posto e' stato preso da un bit di valore **1**; il numero **n** quindi si e' trasformato da positivo in

negativo.

Le considerazioni appena esposte ci fanno capire che se intendiamo utilizzare **SHL** o **SAL** per calcolare rapidamente il prodotto di un numero binario per una potenza intera del **2**, dobbiamo essere certi della validita' del risultato che verra' fornito da queste istruzioni; in sostanza, dobbiamo verificare che nella parte sinistra del numero **n** ci sia un certo "spazio di sicurezza". Nel caso dei numeri senza segno dobbiamo verificare che nella parte sinistra di **n** ci sia un numero sufficiente di zeri; nel caso dei numeri con segno dobbiamo verificare che nella parte sinistra di **n** ci sia un numero sufficiente di bit di segno (o tutti **0** o tutti **1**).

6.8.2 Divisione di un numero intero binario per una potenza intera del 2

Passando alla divisione sappiamo che in base **10**, dividendo un numero intero positivo **n** per **10**, si provoca uno scorrimento di una posizione verso destra delle cifre di **n**; la cifra meno significativa di **n** (cioe' quella piu' a destra) in seguito allo scorrimento, "trabocca" da destra e viene persa. Tutte le cifre di **n** scorrono verso destra di una posizione riducendo quindi il loro peso; infatti la divisione di **n** per **10** trasforma le decine di **n** in unita', le centinaia in decine, le migliaia in centinaia e cosi' via.
La cifra che trabocca da destra rappresenta il resto della divisione; abbiamo ad esempio:
`138 / 10 = Q = 13, R = 8`
Quindi le tre cifre di **138** scorrono verso destra di una posizione facendo uscire l'**8** che rappresenta proprio il resto della divisione.
Analogamente:
`138 / 100 = Q = 1, R = 38`
Le tre cifre di **138** quindi scorrono verso destra di due posizioni facendo uscire il **38** che anche in questo caso, rappresenta il resto della divisione.
Generalizzando, nel caso di una base **b** qualunque, dividendo un numero intero positivo **n** per $\mathbf{b^m}$ si provoca uno scorrimento di **m** posizioni verso destra delle cifre di **n** facendo cosi' traboccare le **m** cifre piu' a destra; le cifre rimaste rappresentano il quoziente, mentre le **m** cifre traboccate da destra rappresentano il resto.
In definitiva, se il divisore puo' essere espresso sotto forma di potenza intera della base, utilizzando questa tecnica si esegue la divisione in modo nettamente piu' rapido rispetto al metodo tradizionale; proprio per questo motivo, abbiamo gia' visto che le **CPU** della famiglia **80x86** forniscono l'istruzione **SHR** (**Shift Logical Right**) per gli scorrimenti verso destra dei bit di un numero intero senza segno, e l'istruzione **SAR** (**Shift Arithmetic Right**) per gli scorrimenti verso destra dei bit di un numero intero con segno. Questa volta pero' le due istruzioni **SHR** e **SAR** non sono equivalenti; la differenza di comportamento tra **SHR** e **SAR** e' legata al fatto che nello scorrimento verso destra dei bit di un numero, e' necessario distinguere tra numeri con segno e numeri senza segno.
Osserviamo infatti che:
`+138 / 10 = Q = +13, R = +8`, mentre `-138 / 10 = Q = -13, R = -8`
Vediamo allora come viene gestita questa situazione da **SHR** e da **SAR**; a tale proposito consideriamo la situazione di Figura 23 che rappresenta come al solito un numero binario contenuto in una locazione da **8** bit:

Figura 23
1 0 1 1 0 0 0 0

Nell'insieme dei numeri interi senza segno, il numero binario **10110000b** equivale a **176**; utilizziamo allora **SHR** per far scorrere ripetutamente di un posto verso destra i bit del numero binario di Figura 23. In questo modo otteniamo la seguente successione:
`01011000b, 00101100b, 00010110b, 00001011b, 00000101b, ...`
Il primo scorrimento produce il risultato **88** (con resto **0b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per **2**; il secondo scorrimento produce il risultato **44** (con resto **00b**)

che e' corretto in quanto rappresenta il quoziente della divisione di **176** per 2^2. Il terzo scorrimento produce il risultato **22** (con resto **000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per 2^3; il quarto scorrimento produce il risultato **11** (con resto **0000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per 2^4. Il quinto scorrimento produce il risultato **5** (con resto **10000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per 2^5.
Nel caso generale possiamo dire allora che l'utilizzo di **SHR** per la divisione rapida di un numero intero senza segno per una potenza intera del **2**, produce sempre il risultato corretto; la sequenza di cifre che trabocca da destra rappresenta sempre il resto della divisione appena effettuata.
Osserviamo poi che **SHR** opera sui numeri interi positivi, per cui riempie con degli zeri i posti che si liberano a sinistra in seguito allo scorrimento verso destra dei bit di un numero binario; in questo modo si ottiene sempre un quoziente positivo. E' chiaro allora che dopo un certo numero di scorrimenti verso destra dei bit di un numero positivo **n**, il quoziente diventa **0**; nel caso ad esempio di un numero binario a **8** bit, dopo **8** scorrimenti verso destra delle cifre di questo numero si ottiene sicuramente un quoziente nullo.

Nell'insieme dei numeri interi con segno, il numero binario **10110000b** di Figura 23 equivale a **-80**; utilizziamo allora **SAR** per far scorrere ripetutamente di un posto verso destra i bit di questo numero binario. In questo modo otteniamo la seguente successione:
`11011000b, 11101100b, 11110110b, 11111011b, 11111101b, ...`
Osserviamo subito che **SAR** riempie con degli **1** i posti che si liberano a sinistra del numero **10110000b** in seguito allo scorrimento delle sue cifre verso destra; questo perche' il **MSB** di **10110000b** e' **1**, e quindi **SAR** deve preservare il bit di segno.
Il primo scorrimento produce il risultato **-40** (con resto **0b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per **2**; il secondo scorrimento produce il risultato **-20** (con resto **00b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per 2^2. Il terzo scorrimento produce il risultato **-10** (con resto **000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per 2^3; il quarto scorrimento produce il risultato **-5** (con resto **0000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per 2^4. Il quinto scorrimento produce il risultato **-3** (con resto **10000b**) che e' pero' sbagliato in quanto dividendo **-80** per 2^5 si dovrebbe ottenere il quoziente **-2** e il resto **-16**!
Dall'analisi dei risultati ottenuti con **SHR** e con **SAR** emerge allora che:

***** Nel caso dei numeri interi senza segno, **SHR** produce sempre il quoziente e il resto corretti.
***** Nel caso dei numeri interi con segno `positivi`, **SAR** produce sempre il quoziente e il resto corretti.
***** Nel caso dei numeri interi con segno `negativi`, **SAR** produce il quoziente e il resto corretti solamente quando il resto e' zero; se invece il resto e' diverso da zero, il quoziente ottenuto appare sommato a **-1**.

Per capire il perche' di questa situazione, bisogna tenere presente che la **CPU**, nell'eseguire una divisione intera tra due numeri interi con o senza segno, arrotonda sempre il quoziente verso lo zero; questo comportamento e' corretto in quanto rispetta le regole matematiche della divisione intera. Ad esempio, il quoziente **+4.9** deve essere arrotondato a **+4** e non a **+5**; analogamente, il quoziente **-6.8** deve essere arrotondato a **-6** e non a **-7**!

Con l'istruzione **SHR**, questa regola matematica viene sempre rispettata; cio' e' dovuto al metodo utilizzato dalla **CPU**, per codificare i numeri interi senza segno.
In riferimento, ad esempio, ai numeri interi senza segno a **8** bit, sappiamo che la **CPU** utilizza le codifiche binarie comprese tra **00000000b** e **11111111b**, per rappresentare i numeri interi senza segno compresi, rispettivamente, tra **0** e **255**; come si puo' notare, i numeri interi senza segno

crescenti (**0**, **1**, **2**, etc) hanno una codifica binaria crescente (**00000000b**, **00000001b**, **00000010b**, etc).
La conseguenza pratica di tutto cio' e' data dal fatto che, utilizzando l'istruzione **SHR** per far scorrere di **n** posti verso destra le cifre di un numero intero senza segno **m**, si ottiene sempre un risultato arrotondato verso lo zero; tale risultato, coincide quindi con il quoziente della divisione intera tra **m** e $\mathbf{2^n}$, e inoltre, le **n** cifre traboccate da destra, coincidono con il resto della divisione stessa!

Con l'istruzione **SAR**, questa regola matematica non viene sempre rispettata; cio' e' dovuto al metodo utilizzato dalla **CPU**, per codificare i numeri interi con segno (codifica binaria in complemento a **2**).
In riferimento, ad esempio, ai numeri interi con segno a **8** bit, sappiamo che:
***** le codifiche binarie comprese tra **00000000b** e **01111111b**, rappresentano i numeri interi positivi compresi, rispettivamente, tra **0** e **+127**;
***** le codifiche binarie comprese tra **10000000b** e **11111111b**, rappresentano i numeri interi negativi compresi, rispettivamente, tra **-128** e **-1**.
Come si puo' notare, i numeri interi positivi crescenti (**+0**, **+1**, **+2**, etc), hanno una codifica binaria crescente (**00000000b**, **00000001b**, **00000010b**, etc); invece, i numeri interi negativi crescenti in valore assoluto (**|-1|**, **|-2|**, **|-3|**, etc), hanno una codifica binaria decrescente (**11111111b**, **11111110b**, **11111101b**, etc)!
La conseguenza pratica di tutto cio' e' data dal fatto che, utilizzando l'istruzione **SAR** per far scorrere di **n** posti verso destra le cifre di un numero intero positivo **m**, si ottiene sempre un risultato arrotondato verso lo zero; tale risultato, coincide quindi con il quoziente della divisione intera tra **m** e $\mathbf{2^n}$, e inoltre, le **n** cifre traboccate da destra, coincidono con il resto della divisione stessa.
Utilizzando l'istruzione **SAR** per far scorrere di **n** posti verso destra le cifre di un numero intero negativo **m**, si ottiene sempre un risultato arrotondato verso l'**infinito negativo**; ad esempio, il risultato **-5.3** viene arrotondato a **-6** anziche' a **-5**.
Se **m** e' un numero intero negativo multiplo intero di $\mathbf{2^n}$, si ottiene un risultato che coincide con il quoziente della divisione intera tra **m** e $\mathbf{2^n}$; infatti, in questo caso il resto e' zero, e quindi il quoziente non necessita di nessun arrotondamento (quoziente esatto).
Se **m** e' un numero intero negativo non divisibile per $\mathbf{2^n}$, si ottiene un risultato che non coincide con il quoziente della divisione intera tra **m** e $\mathbf{2^n}$; infatti, in questo caso il resto e' diverso da zero, e quindi il quoziente necessita di un arrotondamento. Mentre pero' la divisione arrotonda il suo quoziente verso lo zero, l'istruzione **SAR** arrotonda il suo risultato verso l'infinito negativo (ad esempio, il valore **-9.3** viene arrotondato a **-9** dalla divisione, e a **-10** da **SAR**); in tal caso, il risultato prodotto da **SAR** differisce del valore **-1** rispetto al quoziente della divisione!
I problemi appena illustrati, inducono decisamente ad evitare l'uso degli scorrimenti aritmetici a destra per dividere rapidamente numeri negativi per potenze intere del **2**; appare molto piu' sensato, in questo caso, ricorrere alla divisione tradizionale che comporta pero', come sappiamo, una notevole lentezza di esecuzione.

6.8.3 Verifica del risultato attraverso i flags

Come accade per tutte le altre operazioni logico aritmetiche, la **CPU** utilizza i vari flags per fornire al programmatore tutti i dettagli sul risultato di una operazione appena eseguita; analizziamo innanzi tutto le informazioni che ci vengono fornite attraverso **CF**, **SF** e **OF** in relazione al risultato prodotto dalle istruzioni **SHL**, **SAL**, **SHR** e **SAR** applicate ad un numero binario **n**.
Il flag **CF** contiene sempre l'ultimo bit traboccato da **n** (da destra o da sinistra); se ad esempio facciamo scorrere di un posto verso destra i bit di **01001101b**, si ottiene **CF=1** in quanto il bit traboccato da destra ha valore **1**.
Il flag **SF** contiene sempre il **MSB** del risultato appena ottenuto; se ad esempio facciamo scorrere di un posto verso sinistra i bit di **01001101b**, si ottiene il risultato **10011010b**, e quindi **SF=1** in quanto

il **MSB** ha assunto il valore **1**.
Il flag **OF** si porta a **1** ogni volta che il **MSB** del risultato appena ottenuto cambia da **0** a **1** o da **1** a **0**; se ad esempio facciamo scorrere di un posto verso sinistra i bit di **01001101b**, si ottiene il risultato **10011010b**, e quindi **OF=1** in quanto il valore del **MSB** e' cambiato da **0** a **1**.

6.8.4 Esempio di shifter a 4 bit

Vediamo ora come puo' essere implementata una **R.C.** capace di effettuare lo scorrimento dei bit di un numero binario; la Figura 24 illustra uno "**shifter**" a **4** bit capace di implementare tutte le **4** istruzioni **SHL**, **SAL**, **SHR** e **SAR**:

Figura 24

Indichiamo con **A0**, **A1**, **A2** e **A3** i **4** bit del nibble in ingresso, e con **Y0**, **Y1**, **Y2** e **Y3** i **4** bit del nibble in uscita; abbiamo poi l'uscita **CF** che consente di inviare al **Carry Flag** il bit appena scartato, e infine i seguenti tre ingressi di abilitazione:

SHL che abilita lo scorrimento logico a sinistra (equivalente a **SAL**);
SHR che abilita lo scorrimento logico a destra;
SAR che in congiunzione con **SHR** abilita lo scorrimento aritmetico a destra.

Come si puo' constatare attraverso la Figura 23, si ha inoltre:
`SF = Y3` e `OF = A3 EX-NOR Y3`

Ponendo in Figura 23 **SHL=1**, **SHR=0** e **SAR=0**, si abilita la **R.C.** allo scorrimento logico/aritmetico verso sinistra dei bit del nibble in ingresso; come gia' sappiamo, questa operazione produce effetti identici sia sui numeri senza segno, sia sui numeri con segno (**SHL=SAL**).
Ponendo in Figura 23 **SHL=0**, **SHR=1** e **SAR=0**, si abilita la **R.C.** allo scorrimento logico verso destra dei bit del nibble in ingresso; il nibble in ingresso deve essere quindi un numero senza segno.
Ponendo in Figura 23 **SHL=0**, **SHR=1** e **SAR=1**, si abilita la **R.C.** allo scorrimento aritmetico

verso destra dei bit del nibble in ingresso; il nibble in ingresso deve essere quindi un numero con segno.

In relazione agli scorrimenti dei bit di un numero binario, e' necessario sottolineare un aspetto molto importante; come si puo' facilmente intuire, piu' aumenta il numero di shift da effettuare, piu' aumentano i tempi per l'esecuzione di questa operazione da parte della **CPU**. E' chiaro allora che al crescere del numero di shift da effettuare diminuiscono i vantaggi di questa tecnica rispetto alle moltiplicazioni e divisioni tradizionali.

6.9 Rotazione dei bit di un numero binario

In conclusione di questo capitolo, citiamo anche l'operazione di rotazione dei bit di un numero binario; questa operazione e' simile a quella di scorrimento dei bit, con la differenza pero' che i bit che traboccano da una estremita' dell'operando, rientrano in sequenza dall'altra estremita' dell'operando stesso. Appare evidente che in relazione alla operazione di rotazione dei bit, non ha nessun senso la distinzione tra numeri senza segno e numeri con segno.

Le **CPU** della famiglia **80x86** mettono a disposizione **4** istruzioni per la rotazione dei bit, che sono **ROL**, **RCL**, **ROR**, **RCR**.
L'istruzione **ROL** (**Rotate Left**) fa' scorrere verso sinistra i bit di un numero binario; i bit che traboccano da sinistra rientrano in sequenza da destra. In Figura 23 si puo' simulare l'istruzione **ROL** abilitando l'ingresso **SHL**, e collegando l'uscita **Y0** all'uscita **CF**.
L'istruzione **RCL** (**Rotate Through Carry Left**) fa' scorrere verso sinistra i bit di un numero binario; i bit che traboccano da sinistra finiscono in sequenza nel flag **CF**, mentre i vecchi bit contenuti in **CF** rientrano in sequenza dalla destra del numero binario. In Figura 23 si puo' simulare l'istruzione **RCL** abilitando l'ingresso **SHL** e memorizzando in **Y0** il vecchio contenuto di **CF** (prima di ogni shitf).
L'istruzione **ROR** (**Rotate Right**) fa' scorrere verso destra i bit di un numero binario; i bit che traboccano da destra rientrano in sequenza da sinistra. In Figura 23 si puo' simulare l'istruzione **ROR** abilitando l'ingresso **SHR**, e collegando l'uscita **Y3** all'uscita **CF**.
L'istruzione **RCR** (**Rotate Through Carry Right**) fa' scorrere verso destra i bit di un numero binario; i bit che traboccano da destra finiscono in sequenza nel flag **CF**, mentre i vecchi bit contenuti in **CF** rientrano in sequenza dalla sinistra del numero binario. In Figura 23 si puo' simulare l'istruzione **RCR** abilitando l'ingresso **SHR** e memorizzando in **Y3** il vecchio contenuto di **CF** (prima di ogni shitf).

Dalle considerazioni appena esposte si intuisce che la **R.C.** di Figura 23 puo' essere facilmente modificata per poter implementare sia le istruzioni di scorrimento, sia le istruzioni di rotazione dei bit; nella pratica si segue proprio questa strada che permette di ottenere un notevole risparmio di porte logiche.

Capitolo 6 - Reti combinatorie per funzioni matematiche

Nel Capitolo 4 abbiamo visto che dal punto di vista degli esseri umani, elaborare delle informazioni codificate sotto forma di numeri binari significa eseguire su questi numeri una serie di operazioni matematiche che devono produrre dei risultati sempre in formato binario; in base allora a quanto e' stato esposto nel Capitolo 5, possiamo dire che dal punto di vista del computer, elaborare informazioni codificate sotto forma di segnali logici significa eseguire su questi segnali una serie di operazioni che devono produrre dei risultati sempre sotto forma di segnali logici. In questo capitolo vengono analizzate una serie di **R.C.** che permettono alla **CPU** di eseguire elaborazioni sui segnali logici; per noi esseri umani queste elaborazioni effettuate dal computer assumono proprio l'aspetto di operazioni matematiche eseguite sui numeri binari.

6.1 Circuiti Multiplexer e Demultiplexer

Prima di illustrare le **R.C.** per le funzioni matematiche, analizziamo due circuiti che in elettronica digitale trovano un impiego veramente massiccio; si tratta dei circuiti chiamati **Multiplexer** e **Demultiplexer**.
Il **Multiplexer** o **MUX** rappresenta un vero e proprio commutatore elettronico; questo circuito infatti riceve in ingresso **n** segnali logici, ed e' in grado di selezionare in uscita uno solo di essi.
Osserviamo subito che per selezionare una tra **n** possibili linee in ingresso, abbiamo bisogno di **m** linee di selezione con:
$2^m = n$
Da questa relazione si ricava:
$\log_2 2^m = \log_2 n$, e quindi $m = \log_2 n$
Nel caso ad esempio di un **MUX** a **n=4** ingressi, abbiamo bisogno di **m=2** linee di selezione; infatti, con **2** bit possiamo gestire 2^2**=4** configurazioni diverse (**4** numeri binari differenti).
Se colleghiamo ora i **4** ingressi del **MUX** a **4** porte **AND**, grazie alle **2** linee di selezione possiamo abilitare solo una delle porte **AND**; di conseguenza, la porta **AND** abilitata trasferisce in uscita l'ingresso ad essa collegato. Le **4** uscite delle porte **AND** vengono collegate ad una porta **OR** a **4** ingressi per ottenere il segnale che abbiamo selezionato; la Figura 1 illustra la struttura della **R.C.** che implementa un **MUX** a **4** ingressi e una uscita (chiamato anche **MUX** da **4** a **1**):

Figura 1

L'ingresso **E** (**Enable**) permette di attivare o disattivare l'intero **MUX**; portando l'ingresso **E** a livello logico **1**, in uscita dal **NOT** otteniamo un livello logico **0** che disattiva tutte le **4** porte **AND**. In questo modo, l'uscita **Y** del **MUX** si trova sempre a livello logico basso indipendentemente dai **4** segnali in ingresso; se invece **E=0**, il **MUX** viene attivato, ed e' in grado di selezionare uno solo tra i **4** ingressi **a0**, **a1**, **a2**, **a3**.
Le **2** linee **S0** e **S1** rappresentano gli ingressi di selezione; ponendo ora **E=0** (**MUX** attivo), si puo' constatare dalla Figura 1 che:

***** Se **S0 = 0** e **S1 = 0** si ottiene **Y = a0**.
***** Se **S0 = 1** e **S1 = 0** si ottiene **Y = a1**.
***** Se **S0 = 0** e **S1 = 1** si ottiene **Y = a2**.
***** Se **S0 = 1** e **S1 = 1** si ottiene **Y = a3**.

Il compito opposto rispetto al caso del **MUX** viene svolto da un altro circuito chiamato **Demultiplexer** o **DMUX**; il **DMUX** riceve in ingresso un unico segnale logico, ed e' in grado di trasferirlo su una sola tra le **n** linee di uscita. Come al solito, per la selezione delle **n** linee di uscita abbiamo bisogno di:
$m = \log_2 n$ linee di selezione.

Attraverso queste **m** linee di selezione possiamo attivare solo una delle **n** linee di uscita, disattivando tutte le altre **n-1**; si presenta allora il problema di stabilire il livello logico da assegnare alle **n-1** linee di uscita disabilitate. E' chiaro che se vogliamo trasferire sull'uscita abilitata un segnale a livello logico **1**, dobbiamo portare a livello logico **0** tutte le **n-1** uscite disabilitate; viceversa, se vogliamo trasferire sull'uscita abilitata un segnale a livello logico **0**, dobbiamo portare a livello logico **1** tutte le **n-1** uscite disabilitate.

Per ottenere queste caratteristiche, gli **m** ingressi di selezione agiscono su **n** porte **NAND** ciascuna delle quali fornisce una delle **n** uscite; la Figura 2 illustra la struttura della **R.C.** che implementa un **DMUX** a **1** ingresso e **4** uscite (chiamato anche **DMUX** da **1** a **4**):

Figura 2

In questo circuito, **S0** e **S1** sono gli ingressi di selezione, **D** e' l'unico ingresso per i dati, mentre **E** e' come al solito l'ingresso di abilitazione; le **4** uscite del **DMUX** sono **Y0**, **Y1**, **Y2** e **Y3**.
Il **DMUX** di Figura 2 permette di trasferire su una delle **4** uscite un segnale a livello logico **0**; di

conseguenza, le **3** uscite disabilitate si portano a livello logico **1**. Per ottenere questo trasferimento bisogna porre **E=0** e **D=1**; in queste condizioni possiamo constatare che:

***** Se **S0 = 0** e **S1 = 0** si ottiene **Y0 = 0**, **Y1 = 1**, **Y2 = 1**, **Y3 = 1**.
***** Se **S0 = 1** e **S1 = 0** si ottiene **Y0 = 1**, **Y1 = 0**, **Y2 = 1**, **Y3 = 1**.
***** Se **S0 = 0** e **S1 = 1** si ottiene **Y0 = 1**, **Y1 = 1**, **Y2 = 0**, **Y3 = 1**.
***** Se **S0 = 1** e **S1 = 1** si ottiene **Y0 = 1**, **Y1 = 1**, **Y2 = 1**, **Y3 = 0**.

6.2 Comparazione tra numeri binari

Tra le istruzioni piu' importanti messe a disposizione dai linguaggi di programmazione di alto livello, troviamo sicuramente quelle denominate **istruzioni per il controllo del flusso**, cosi' chiamate perche' consentono ad un programma di prendere delle decisioni in base al risultato di una operazione appena eseguita; si possono citare ad esempio istruzioni come **IF**, **THEN**, **ELSE**, **DO WHILE**, **REPEAT UNTIL**, etc. Dal punto di vista della **CPU** tutte queste istruzioni si traducono in una serie di confronti tra numeri binari; proprio per questo motivo, tutte le **CPU** sono dotate di apposite **R.C.** che permettono di effettuare comparazioni tra numeri binari.
Supponiamo ad esempio di avere due numeri binari **A** e **B** a **8** bit che rappresentano i codici ASCII di due lettere dell'alfabeto; consideriamo ora un programma che elabora questi due numeri attraverso le pseudo-istruzioni mostrate in Figura 3:

Figura 3

```
SE (A e' uguale a B)
   esegui la Procedura 1
ALTRIMENTI
   esegui la Procedura 2
```

Come si puo' notare, questa porzione del programma effettua una scelta basata sul risultato del confronto tra **A** e **B**; se **A** e' uguale a **B**, l'esecuzione salta al blocco di istruzioni denominato **Procedura 1**, mentre in caso contrario (**A** diverso da **B**), l'esecuzione salta al blocco di istruzioni denominato **Procedura 2**.
Per comparare **A** con **B**, abbiamo bisogno di una **funzione booleana** che riceve in input i due numeri binari da confrontare e restituisce in output un risultato espresso sempre in forma binaria; possiamo usare ad esempio il valore **0** per indicare che i due numeri sono diversi tra loro, e il valore **1** per indicare che i due numeri sono uguali. Una volta scritta la funzione booleana, si puo' passare alla sua realizzazione pratica tramite una **R.C.**.
Nel precedente capitolo, abbiamo visto che una porta **EX-OR** a due ingressi restituisce **0** se i due livelli logici in ingresso sono uguali tra loro, e restituisce invece **1** in caso contrario; analogamente, una porta **EX-NOR** a due ingressi restituisce **1** se i due livelli logici in ingresso sono uguali tra loro, e restituisce invece **0** in caso contrario. Le porte **EX-OR** oppure le porte **EX-NOR** appaiono quindi particolarmente adatte a risolvere il nostro problema; per capire come si debba effettuare il confronto tra due numeri binari, bisogna fare innanzi tutto alcune considerazioni importanti:

***** Il confronto ha senso solo se i due numeri binari hanno lo stesso numero di bit.
***** Due numeri binari sono uguali quando hanno i bit corrispondenti (di pari posizione) uguali tra loro; quindi il bit in posizione **0** del numero **A** deve essere uguale al bit in posizione **0** del numero **B**, il bit in posizione **1** del numero **A** deve essere uguale al bit in posizione **1** del numero **B** e cosi' via.

In base a queste considerazioni, vediamo subito come si deve procedere utilizzando ad esempio le porte **EX-NOR** che restituiscono **1** se i due livelli logici in ingresso sono uguali tra loro, e **0** se sono

diversi; utilizziamo una porta **EX-NOR** per ogni coppia di bit (corrispondenti) da confrontare. La prima porta confronta il bit in posizione **0** del primo numero con il bit in posizione **0** del secondo numero; la seconda porta confronta il bit in posizione **1** del primo numero con il bit in posizione **1** del secondo numero e cosi' via. Ogni porta **EX-NOR** produce un risultato che vale **1** se i due bit sono uguali tra loro, e vale invece **0** in caso contrario; i due numeri binari sono uguali tra loro solo quando tutte le porte **EX-NOR** producono **1** come risultato, cioe' quando tutti i bit corrispondenti dei due numeri sono uguali tra loro. Se almeno una porta **EX-NOR** produce **0** come risultato, i due numeri binari sono diversi tra loro.
Per sapere se tutte le porte **EX-NOR** hanno prodotto **1**, basta confrontare con un **AND** tutte le loro uscite; l'**AND** ci restituisce **1** solo se tutti i suoi ingressi valgono **1**, e ci restituisce **0** se anche un solo ingresso vale **0**. Facendo riferimento per semplicita' a numeri a **4** bit (nibble), la nostra funzione booleana sara' allora:

```
Y = (A0 EX-NOR B0) AND (A1 EX-NOR B1) AND (A2 EX-NOR B2) AND (A3 EX-NOR B3)
```

dove **A0**, **A1**, **A2**, **A3** rappresentano i **4** bit del primo numero e **B0**, **B1**, **B2**, **B3** rappresentano i **4** bit del secondo numero.
La Figura 4 mostra la **R.C** che implementa in pratica questa funzione:

Figura 4

Comparatore a 4 bit

Si capisce subito che il numero di porte **EX-NOR** da utilizzare e' pari al numero di coppie di bit da confrontare; per confrontare ad esempio due numeri a **8** bit (**8** coppie di bit) occorreranno **8** porte **EX-NOR**. Le uscite di tutte le porte **EX-NOR** impiegate, vanno collegate agli ingressi di una porta **AND** che restituisce **1** solo quando tutti i suoi ingressi valgono **1**; ma questo accade solo quando tutti i confronti hanno prodotto **1** come risultato (bit corrispondenti uguali tra loro). In definitiva, la **R.C.** di Figura 4 fornisce in output il livello logico **1** se i due numeri da confrontare sono uguali tra loro, e il livello logico **0** in caso contrario.
Come si puo' notare, ogni porta **EX-NOR** confronta un bit del primo numero con il bit che nel secondo numero occupa la stessa posizione; questo tipo di confronto viene definito **bit a bit** (in inglese **bitwise**).
Tutti i concetti appena esposti, si estendono immediatamente al confronto tra numeri binari di qualunque dimensione; se vogliamo confrontare numeri binari a **16** bit, ci occorreranno **16** porte **EX-NOR** e una porta **AND** a **16** ingressi; se vogliamo confrontare numeri binari a **32** bit ci occorreranno **32** porte **EX-NOR** e una porta **AND** a **32** ingressi e cosi' via. Queste considerazioni comunque sono solamente teoriche perche' in realta', nella realizzazione pratica dei comparatori (e di altre **R.C.**) si segue un'altra strada; puo' capitare infatti che in commercio si trovino solo i comparatori a **4** bit illustrati in Figura 4. Come si puo' facilmente intuire, in questo caso basta utilizzare uno o piu' comparatori a **4** bit collegati in parallelo. Se ad esempio vogliamo confrontare numeri a **16** bit, utilizzeremo **4** comparatori a **4** bit in parallelo (**4x4=16**); se invece vogliamo confrontare numeri a **32** bit, utilizzeremo **8** comparatori a **4** bit in parallelo (**8x4=32**) e cosi' via.

Ogni comparatore ha una sola uscita e tutte queste uscite vanno collegate agli ingressi di una porta **AND** che produrra' il risultato finale. Si ricorda che le porte **AND** con piu' di due ingressi, producono in uscita un livello logico alto solo se tutti i livelli logici in ingresso sono alti; in caso contrario verra' prodotto un livello logico basso.

Un caso che si presenta molto piu' di frequente consiste nel voler conoscere la relazione esatta che esiste tra due numeri binari; dati cioe' i due numeri binari **A** e **B**, vogliamo sapere se **A** e' minore di **B**, o se **A** e' uguale a **B**, o se **A** e' maggiore di **B**. In Figura 5 vediamo un esempio relativo sempre al confronto tra nibble:

Figura 5

Prima di tutto osserviamo in Figura 5a la struttura elementare del circuito che confronta il bit **Ai** del numero **A** con il corrispondente bit **Bi** del numero **B**; come si puo' facilmente constatare, in base al risultato del confronto solo una delle tre uscite sara' a livello logico alto, mentre le altre due saranno a livello logico basso. In particolare notiamo che la prima uscita (quella piu' in alto) presenta un livello logico alto solo se **Ai** e' maggiore di **Bi** (cioe' **Ai=1** e **Bi=0**); la seconda uscita presenta un livello logico alto solo se **Ai** e' uguale a **Bi** (cioe' **Ai=0** e **Bi=0**, oppure **Ai=1** e **Bi=1**). La terza uscita infine presenta un livello logico alto solo se **Ai** e' minore di **Bi** (cioe' **Ai=0** e **Bi=1**).
Partendo dal circuito elementare di Figura 5a si puo' facilmente ottenere un comparatore completo a **4** bit che ci permette di sapere quale relazione esiste tra il nibble **A** e il nibble **B**; il blocco che rappresenta simbolicamente questo comparatore, viene mostrato in Figura 5b. I comparatori a **4** bit che si trovano in commercio, sono dotati come si puo' notare di **3** uscite indicate con **Y1**, **Y2** e **Y3**; l'uscita **Y1** e' associata alla condizione **A** maggiore di **B**, l'uscita **Y2** e' associata alla condizione **A** uguale a **B** e l'uscita **Y3** e' associata alla condizione **A** minore di **B**.
Possiamo dire quindi che se **A** e' maggiore di **B** si ottiene:
`Y1 = 1, Y2 = 0, Y3 = 0`
Se **A** e' uguale a **B** si ottiene:
`Y1 = 0, Y2 = 1, Y3 = 0`
Se **A** e' minore di **B** si ottiene:
`Y1 = 0, Y2 = 0, Y3 = 1`

La tecnica che si utilizza per il confronto e' molto semplice e si basa su una proprieta' fondamentale dei sistemi di numerazione posizionali; come sappiamo infatti, i numeri rappresentati con questo

sistema, sono formati da una sequenza di cifre il cui peso dipende dalla posizione della cifra stessa nel numero. La cifra di peso maggiore e' quella piu' a sinistra e quindi, per confrontare due numeri nel modo piu' rapido possibile, e' necessario partire dal confronto tra le cifre piu' significative dei due numeri stessi (**MSB** per i numeri binari); ancora una volta bisogna ricordare che l'operazione di confronto ha senso solo se i due numeri hanno lo stesso numero di cifre.
Tornando allora al caso del comparatore a **4** bit di Figura 5b, vediamo come si svolge il confronto tra i due nibble **A** e **B**:

1) Se il bit **A3** e' maggiore del bit **B3** e' inutile continuare perche' sicuramente il nibble **A** e' maggiore del nibble **B**; di conseguenza **Y1** sara' a livello logico **1** mentre le altre due uscite saranno a livello logico **0**.

2) Se il bit **A3** e' minore del bit **B3** e' inutile continuare perche' sicuramente il nibble **A** e' minore del nibble **B**; di conseguenza **Y3** sara' a livello logico **1** mentre le altre due uscite saranno a livello logico **0**.

3) Se il bit **A3** e' uguale al bit **B3** si deve necessariamente passare ai bit in posizione **2** ripetendo i passi **1** e **2** appena descritti; se anche il bit **A2** e' uguale al bit **B2**, si passa ai bit in posizione **1** e cosi' via, sino ad arrivare eventualmente ai bit in posizione **0** (**LSB**). Solo dopo quest'ultimo confronto possiamo dire con certezza se i due numeri sono uguali tra loro e in questo caso avremo **Y2** a livello logico **1** e le altre uscite a livello logico **0**.

Nella Figura 5b si nota anche la presenza di tre connessioni **I1**, **I2** e **I3**; queste connessioni servono per il collegamento in parallelo di due o piu' comparatori quando si ha la necessita' di confrontare tra loro numeri con piu' di **4** bit.
La Figura 6 illustra un esempio che si riferisce al confronto tra numeri binari a **8** bit; vengono utilizzati a tale proposito due comparatori a **4** bit, e cioe' **C1** che confronta il nibble basso e **C2** che confronta il nibble alto.

Figura 6

Comparazione tra numeri a 8 bit

Il confronto parte come al solito dai bit piu' significativi che in questo caso si trovano in posizione **7**; indichiamo con **Aa** e **Ba** i nibble alti, e con **Ab** e **Bb** i nibble bassi dei due numeri **A** e **B**. Se il comparatore **C2** confrontando i nibble alti verifica che **Aa** e' maggiore di **Ba** o che **Aa** e' minore di **Ba**, non ha bisogno di consultare **C1**; in questo caso **C2** ignora gli ingressi **I1**, **I2** e **I3**, e termina il confronto fornendo i risultati sulle sue uscite **Y1**, **Y2** e **Y3**.
Se invece il comparatore **C2** verifica che **Aa** e' uguale a **Ba**, allora legge i suoi ingressi **I1**, **I2**, **I3** che gli forniscono il risultato del confronto tra i nibble bassi effettuato da **C1**; come si puo' notare dalla Figura 6, l'ingresso **I1** e' associato a **Ab** maggiore di **Bb**, l'ingresso **I2** e' associato a **Ab** uguale a **Bb** e l'ingresso **I3** e' associato a **Ab** minore di **Bb**. In base ai livelli logici assunti da **I1**, **I2** e **I3**, il comparatore **C2** fornisce il risultato finale sempre attraverso le sue uscite **Y1**, **Y2**, **Y3**; anche in questo caso si nota che la condizione **A** uguale a **B** puo' essere verificata con certezza solo dopo l'ultimo confronto che coinvolge i bit **A0** e **B0** (cioe' gli **LSB**).
Naturalmente, nell'esempio di Figura 6, gli ingressi **I1**, **I2** e **I3** del comparatore **C1** non vengono utilizzati; appare anche evidente il fatto che attraverso il collegamento in parallelo di piu' comparatori a **4** bit, si possono confrontare tra loro numeri binari di qualunque dimensione.
Un computer con architettura a **8** bit, sara' dotato di comparatori a **8** bit; un computer con architettura a **16** bit sara' dotato di comparatori a **16** bit e cosi' via. Quindi su un computer con architettura a **8** bit, non sara' possibile confrontare via hardware numeri a **16** bit; se si vuole effettuare ugualmente il confronto tra numeri a **16** bit si dovra' procedere via software scrivendo un apposito programma che naturalmente non potra' raggiungere le prestazioni del confronto via hardware. Molti linguaggi di programmazione di alto livello, mettono a disposizione delle funzioni (sottoprogrammi) che permettono di effettuare via software diverse operazioni su numeri formati da un numero di bit maggiore di quello dell'architettura del computer che si sta utilizzando.

6.3 Addizione tra numeri binari

Come abbiamo visto nei precedenti capitoli, per sommare due numeri espressi nel sistema posizionale arabo, bisogna incolonnarli in modo da far corrispondere sulla stessa colonna le cifre aventi peso identico nei due numeri stessi; sappiamo anche che la somma viene eseguita a partire dalla colonna piu' a destra che contiene le cifre meno significative dei due addendi. Questa prima somma non deve tenere conto di nessun riporto precedente; nel sommare invece le colonne successive alla prima, si dovra' tenere conto dell'eventuale riporto proveniente dalla somma della colonna precedente.
E' importante ricordare inoltre che in binario, nel sommare la prima colonna, la situazione piu' critica che si puo' presentare e':
`1 + 1`
che da' come risultato **0** con riporto di **1**; nel sommare le colonne successive alla prima, in presenza di un riporto pari a **1**, la situazione piu' critica che si puo' presentare e':
`1 + 1 + 1`
che da' come risultato **1** con riporto di **1**. In base **2** quindi e in qualsiasi altra base, il riporto massimo non puo' superare **1**.
Da quanto abbiamo appena detto, si puo' capire che la **R.C.** che dovra' eseguire la somma della prima colonna, sara' molto piu' semplice della **R.C.** che somma le colonne successive tenendo conto di eventuali riporti; per questo motivo, i circuiti che sommano le coppie di bit si suddividono in due tipi chiamati: **Half Adder** e **Full Adder**.

La Figura 7a illustra la tabella della verita' di un **Half Adder** (**H.A.**) o semisommatore binario; la Figura 7b illustra la **R.C.** che permette di implementare la somma di due bit.

Figura 7

Input		Output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

L'**H.A.** esegue la somma relativa ai due bit che nell'addizione occupano la prima colonna (**LSB**), e produce quindi un valore a **2** bit compreso tra **00b** e **10b**; nella tabella della verita' indichiamo con **S** il bit piu' a destra, e con **C** il bit piu' a sinistra della somma appena effettuata dall'**H.A.**. Il bit **S** rappresenta una delle cifre del risultato finale (si tratta per la precisione del **LSB**); il bit **C** (**Carry**) rappresenta il riporto (**0** o **1**) di cui si dovra' tenere conto nella somma relativa alla colonna successiva dell'addizione.
Osservando la tabella della verita' dell'**H.A.** in Figura 7a, si vede subito che il contenuto della colonna **S** coincide esattamente con i livelli logici in uscita da una porta **EX-OR** in seguito alla operazione:
`A EX-OR B`
Notiamo anche che il contenuto della colonna **C** (riporto destinato alla colonna successiva), coincide esattamente con i livelli logici forniti in uscita da una porta **AND** in seguito alla operazione:
`A AND B`
In base a questi risultati, siamo in grado di definire le due funzioni che caratterizzano l'**H.A.** e che sono:
`S = A EX-OR B` e `C = A AND B`
La Figura 7b illustra la **R.C.** che soddisfa la tabella della verita' di Figura 7a; possiamo dire quindi che l'**H.A.** ha bisogno di due ingressi che rappresentano i due bit da sommare, e di due uscite che rappresentano una delle cifre del risultato finale e il riporto destinato alla colonna successiva dell'addizione.
La Figura 8a illustra la tabella della verita' di un **Full Adder** (**F.A.**) o sommatore binario; la Figura 8b illustra la **R.C.** che permette di implementare la somma di due bit piu' il riporto.

Figura 8

Input			Output	
C0	A1	B1	S1	C1
0	0	0	0	0
1	0	0	1	0
0	0	1	1	0
1	0	1	0	1
0	1	0	1	0
1	1	0	0	1
0	1	1	0	1
1	1	1	1	1

Il **F.A.** esegue la somma relativa ai due bit che nell'addizione occupano una tra le colonne successive alla prima, e deve tenere conto quindi anche del bit di riporto proveniente dalla colonna precedente; questa volta la somma produce un valore a **2** bit che sara' compreso tra **00b** e **11b**. Nella tabella della verita' indichiamo con **S** il bit piu' a destra, e con **C** il bit piu' a sinistra della somma appena effettuata dal **F.A.**; il bit **S** rappresenta una delle cifre del risultato finale, mentre il bit **C** (**Carry**) rappresenta il riporto (**0** o **1**) destinato alla colonna successiva dell'addizione.
Supponiamo di dover sommare i bit della seconda colonna di una addizione, e indichiamo con **C0** il riporto precedente, con **A1** e **B1** i due bit da sommare, con **S1** la cifra meno significativa della somma **A1+B1** e con **C1** il riporto di **A1+B1** destinato alla colonna successiva; la tabella della verita' di Figura 8a e' formata da **8** righe in quanto per ciascuno dei **4** possibili risultati della somma **A1+B1**, dobbiamo considerare i due casi **C0=0** e **C0=1** per un totale di **4x2=8** casi distinti.
Le funzioni booleane che realizzano il **F.A.** possono essere ottenute facilmente ricordando la tecnica che si utilizza per eseguire le addizioni con carta e penna; in relazione alle colonne successive alla prima, sappiamo che prima di tutto si sommano le due cifre in colonna, dopo di che si aggiunge al risultato ottenuto il riporto precedente.
Queste due addizioni consecutive suggeriscono l'idea di utilizzare per il **F.A.**, due **H.A.** in serie come si vede in Figura 8b; il primo **H.A.** somma i due bit in colonna (**A1+B1**) producendo in uscita i risultati parziali **S'** e **C'**, mentre il secondo **H.A.** somma **S'** con il riporto precedente **C0** producendo a sua volta i risultati parziali **S''** e **C''**. Alla fine **S''** rappresenta la cifra **S1** del risultato finale, mentre i due riporti **C'** e **C''** devono fornirci il riporto **C1** destinato alla colonna successiva.
Come si verifica facilmente dal circuito di Figura 8b, i due riporti **C'** e **C''** non possono mai essere entrambi a livello logico **1** (la somma massima infatti e' **11b**), e quindi basta applicarli ai due ingressi di una porta **OR** per ottenere il riporto **C1** destinato alla colonna successiva.
Possiamo dire quindi che il **F.A.** ha bisogno di tre ingressi che rappresentano i due bit da sommare e il riporto proveniente dalla colonna precedente; le due uscite del **F.A.** forniscono invece una delle cifre del risultato finale e il riporto destinato alla colonna successiva.

Una volta realizzati l'**H.A.** e il **F.A.**, possiamo realizzare circuiti per sommare numeri binari di qualsiasi dimensione; lo schema a blocchi di Figura 9 mostra una **R.C.** per la somma tra numeri binari a **4** bit:

Figura 9

La **R.C.** di Figura 9 utilizza un **H.A.** e tre **F.A.**; l'**H.A.** somma i bit della prima colonna (**LSB**), mentre i tre **F.A.** sommano i bit delle tre colonne successive. Il riporto finale **C3** contiene il valore (**0** o **1**) che verra' assegnato al **Carry Flag**; come gia' sappiamo, grazie a **CF** possiamo sommare via software numeri binari di qualsiasi ampiezza in bit.
Anche nel caso dell'addizione (e di tutte le altre operazioni), bisogna ricordare che un computer con architettura ad esempio a **16** bit, sara' dotato di circuiti in grado di sommare via hardware numeri a **16** bit; se vogliamo sommare tra loro numeri a **32** bit, dobbiamo procedere via software suddividendo i bit degli addendi in gruppi da **16**.
Nei precedenti capitoli abbiamo anche appurato che grazie all'aritmetica modulare, una **R.C.** come quella di Figura 9 puo' sommare indifferentemente numeri con o senza segno; alla fine dell'addizione, se stiamo operando sui numeri senza segno dobbiamo consultare **CF** per conoscere il

riporto finale, mentre se stiamo operando sui numeri con segno dobbiamo consultare **OF** e **SF** per sapere se il risultato e' valido e per individuarne il segno. Come si puo' facilmente verificare attraverso la Figura 9, si ha:

`CF = C3`, `SF = S3` e `OF = (A3 EX-NOR B3) AND (A3 EX-OR B3 EX-OR S3)`

6.4 Sottrazione tra numeri binari

Per la sottrazione valgono considerazioni analoghe a quelle svolte per l'addizione; sottraendo i bit della colonna piu' a destra non si deve tenere conto di eventuali prestiti chiesti dalle colonne precedenti; sottraendo invece i bit delle colonne successive alla prima, bisogna tenere conto degli eventuali prestiti chiesti dalle colonne precedenti. Nel primo caso, la situazione piu' critica che puo' presentarsi e':

`0 - 1`

che da' come risultato **1** con prestito di **1** dalle colonne successive; nel secondo caso, in presenza di un prestito pari a **1** richiesto dalla colonna precedente, la situazione piu' critica che puo' presentarsi e':

`0 - 1 - 1`

che da' come risultato **0** con prestito di **1** dalle colonne successive.
In qualsiasi base numerica quindi, il prestito massimo non puo' superare **1**; per gestire questi due casi possibili vengono impiegati due circuiti chiamati **Half Subtractor** e **Full Subtractor**.

La Figura 10a illustra la tabella della verita' di un **Half Subtractor** (**H.S.**) o semisottrattore binario; la Figura 10b illustra la **R.C.** che permette di implementare la sottrazione tra due bit.

Figura 10

Input		Output	
A	B	D	P
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

L'**H.S.** calcola la differenza tra i due bit che nella sottrazione occupano la prima colonna (**LSB**), e produce quindi come risultato un valore a **1** bit compreso tra **0b** e **1b**; nella tabella della verita' indichiamo questo bit con **D**. Il bit **D** rappresenta una delle cifre del risultato finale (in questo caso si tratta del **LSB**); indichiamo poi con **P** il bit che rappresenta il prestito (**0** o **1**) richiesto alle colonne successive.
Osservando la tabella della verita' dell'**H.S.** in Figura 10a, si vede subito che il contenuto della colonna **D** coincide esattamente con i livelli logici in uscita da una porta **EX-OR** in seguito alla operazione:

`A EX-OR B`

Notiamo anche che il contenuto della colonna **P** (prestito richiesto alle colonne successive), coincide esattamente con i livelli logici forniti in uscita da una porta **AND** in seguito alla operazione:

`NOT(A) AND B`

In base a questi risultati, siamo in grado di definire le due funzioni che caratterizzano l'**H.S.** e che sono:

`D = A EX-OR B` e `P = NOT(A) AND B`

La Figura 10b illustra la **R.C.** che soddisfa la tabella della verita' di Figura 10a; possiamo dire

quindi che l'**H.S.** ha bisogno di due ingressi che rappresentano i due bit da sottrarre, e di due uscite che rappresentano una delle cifre del risultato finale (in questo caso si tratta dell'**LSB**) e il prestito richiesto alle colonne successive della sottrazione.

La Figura 11a illustra la tabella della verita' di un **Full Subtractor** (**F.S.**) o sottrattore binario; la Figura 11b illustra la **R.C.** che permette di implementare la differenza tra due bit tenendo conto della presenza di un eventuale prestito.

Figura 11

Input			Output	
P0	A1	B1	D1	P1
0	0	0	0	0
1	0	0	1	1
0	0	1	1	1
1	0	1	0	1
0	1	0	1	0
1	1	0	0	0
0	1	1	0	0
1	1	1	1	1

Il **F.S.** calcola la differenza tra i due bit che nella sottrazione occupano una tra le colonne successive alla prima, e produce quindi come risultato un valore a **1** bit compreso tra **0b** e **1b**; nella tabella della verita' indichiamo questo bit con **D**. Il bit **D** rappresenta una delle cifre del risultato finale; indichiamo poi con **P** il bit che rappresenta il prestito (**0** o **1**) richiesto alle colonne successive.
Supponiamo di dover calcolare la differenza tra i bit della seconda colonna di una sottrazione, e indichiamo con **P0** il prestito precedente, con **A1** e **B1** i due bit da sottrarre, con **D1** la cifra meno significativa della sottrazione **A1-B1** e con **P1** il prestito richiesto da **A1-B1** alle colonne successive; la tabella della verita' di Figura 11a e' formata da **8** righe in quanto per ciascuno dei **4** possibili risultati della differenza **A1-B1**, dobbiamo considerare i due casi **P0=0** e **P0=1** per un totale di **4x2=8** casi distinti.
Anche in questo caso le funzioni booleane che realizzano il **F.S.** possono essere ottenute facilmente ricordando la tecnica che si utilizza per eseguire le sottrazioni con carta e penna; in relazione alle colonne successive alla prima, possiamo iniziare dalla sottrazione tra le due cifre in colonna, dopo di che sottraiamo al risultato ottenuto il prestito precedente.
Queste due sottrazioni consecutive suggeriscono l'idea di utilizzare per il **F.S.**, due **H.S.** in serie come si vede in Figura 11b; il primo **H.S.** sottrae i due bit in colonna (**A1-B1**) producendo in uscita i risultati parziali **D'** e **P'**, mentre il secondo **H.S.** sottrae **D'** con il prestito precedente **P0** producendo a sua volta i risultati parziali **D''** e **P''**. Alla fine **P''** rappresenta la cifra **D1** del risultato finale, mentre i due prestiti **P'** e **P''** devono fornirci il riporto **P1** richiesto alle colonne successive.
Come si verifica facilmente dal circuito di Figura 11b, i due prestiti **P'** e **P''** non possono mai essere entrambi a livello logico **1** (il prestito massimo infatti e' **1**), e quindi basta applicarli ai due ingressi di una porta **OR** per ottenere il prestito **P1** richiesto alle colonne successive.
Possiamo dire quindi che il **F.S.** ha bisogno di tre ingressi che rappresentano i due bit da sottrarre e il prestito richiesto dalla colonna precedente; le due uscite del **F.S.** forniscono invece una delle cifre del risultato finale e il prestito richiesto alle colonne successive.

Una volta realizzati l'**H.S.** e il **F.S.**, possiamo realizzare circuiti per sottrarre numeri binari di qualsiasi dimensione; lo schema a blocchi di Figura 12 mostra una **R.C.** per la sottrazione tra numeri binari a **4** bit:

Figura 12

La **R.C.** di Figura 12 utilizza un **H.S.** e tre **F.S.**; l'**H.S.** sottrae i bit della prima colonna (**LSB**), mentre i tre **F.S.** sottraggono i bit delle tre colonne successive. Il prestito finale **P3** contiene il valore (**0** o **1**) che verra' assegnato al **Carry Flag**; come gia' sappiamo, grazie a **CF** possiamo sottrarre via software numeri binari di qualsiasi ampiezza in bit.
Un computer con architettura ad esempio a **16** bit, sara' dotato di circuiti in grado di sottrarre via hardware numeri a **16** bit; se vogliamo sottrarre tra loro numeri a **32** bit, dobbiamo procedere via software suddividendo i bit del minuendo e del sottraendo in gruppi da **16**.
Nei precedenti capitoli abbiamo anche appurato che grazie all'aritmetica modulare, una **R.C.** come quella di Figura 12 puo' sottrarre indifferentemente numeri con o senza segno; alla fine della sottrazione, se stiamo operando sui numeri senza segno dobbiamo consultare **CF** per conoscere il prestito finale, mentre se stiamo operando sui numeri con segno dobbiamo consultare **OF** e **SF** per sapere se il risultato e' valido e per individuarne il segno. Come si puo' facilmente verificare attraverso la Figura 12, si ha:
`CF = P3`, `SF = D3` e `OF = (A3 EX-NOR B3) AND (A3 EX-OR B3 EX-OR D3)`

6.5 Circuiti complementatori

Nella precedente sezione sono stati descritti i circuiti che eseguono la sottrazione tra numeri binari; nella realizzazione pratica di questi circuiti puo' essere seguita anche un'altra strada. Osserviamo infatti che, dati i due numeri binari **n1** e **n2**, possiamo scrivere:
`n1 - n2 = n1 + (-n2)`
In sostanza, la differenza tra **n1** e **n2** e' pari alla somma tra **n1** e l'opposto di **n2**; come gia' sappiamo, l'opposto di un numero binario si ottiene effettuando il complemento a **2** del numero stesso. Il complemento a **2** (**NEG**) di un numero binario **n** si ottiene invertendo tutti i suoi bit (**NOT**) e sommando **1** al risultato ottenuto; possiamo scrivere quindi:
`NEG(n) = NOT(n) + 1`
Per realizzare nel modo piu' semplice il complemento a **1** di un numero, possiamo utilizzare delle semplici porte **NOT** che come sappiamo producono in uscita un livello logico che e' l'opposto di quello in ingresso; se entra un segnale alto, esce un segnale basso, mentre se entra un segnale basso, esce un segnale alto. Quindi se vogliamo complementare ad esempio numeri binari a **16** bit, possiamo utilizzare **16** porte **NOT**; ciascuna di queste porte inverte uno dei bit del numero da complementare, e alla fine otteniamo in uscita il numero in ingresso con tutti i bit invertiti.
Il circuito complementatore appena descritto, e' molto semplice; nella pratica pero' si segue un'altra strada legata come al solito alla semplificazione circuitale. Invece di realizzare una **R.C.** per ogni funzione booleana, si preferisce realizzare delle **R.C.** leggermente piu' complesse, che permettono pero' di svolgere diverse funzioni con lo stesso circuito; in Figura 13 ad esempio, vediamo un

dispositivo a **4** bit che svolge **4** funzioni differenti a seconda dei livelli logici assunti dai cosiddetti **ingressi di selezione** che sono **S0** e **S1**:

Select		Input				Output			
S0	S1	I0	I1	I2	I3	Y0	Y1	Y2	Y3
0	0	A0	A1	A2	A3	$\overline{A0}$	$\overline{A1}$	$\overline{A2}$	$\overline{A3}$
0	1	A0	A1	A2	A3	A0	A1	A2	A3
1	0	--	--	--	--	1	1	1	1
1	1	--	--	--	--	0	0	0	0

Figura 13

Dalla tabella della verita' di Figura 13 si vede che attribuendo ai due ingressi **S0** e **S1** i **4** (**2^2**) possibili valori binari **00b**, **01b**, **10b** e **11b**, si possono ottenere in uscita **4** differenti elaborazioni dello stesso nibble in ingresso. In particolare:

***** Per **S0 = 0** e **S1 = 0** si ottiene in uscita il complemento a **1** del nibble in ingresso.
***** Per **S0 = 0** e **S1 = 1** si ottiene in uscita lo stesso nibble in ingresso.
***** Per **S0 = 1** e **S1 = 0** si ottiene in uscita il nibble **1111b**.
***** Per **S0 = 1** e **S1 = 1** si ottiene in uscita il nibble **0000b**.

In questi ultimi due casi, il nibble in ingresso e' ininfluente; con lo stesso circuito quindi realizziamo **4** funzioni differenti che altrimenti avrebbero richiesto **4** circuiti diversi.
I circuiti complementatori sono molto importanti in quanto vengono largamente utilizzati dalla **CPU** in molte operazioni; ricordiamo ad esempio che la **CPU** esegue moltiplicazioni e divisioni solo sui numeri senza segno. Eventuali numeri negativi vengono prima di tutto convertiti nei corrispondenti numeri positivi; per poter effettuare queste conversioni vengono appunto utilizzati i circuiti complementatori.

6.6 Moltiplicazione tra numeri binari

Nei precedenti capitoli e' stato detto che la **CPU** effettua la moltiplicazione attraverso un metodo fortemente basato sullo stesso procedimento che utilizziamo noi esseri umani quando eseguiamo questa operazione con carta e penna; abbiamo anche visto che la moltiplicazione provoca un cambiamento di modulo, per cui la **CPU** ha bisogno di sapere se vogliamo operare nell'insieme dei numeri senza segno o nell'insieme dei numeri con segno.
Moltiplicando tra loro due numeri binari a **n** bit, otteniamo un risultato interamente rappresentabile attraverso un numero binario a **2n** bit; partiamo allora dai numeri interi senza segno a **4** bit (nibble), e supponiamo di voler calcolare ad esempio:

```
10 x 13 = 130
```

Traducendo tutto in binario, e applicando lo stesso procedimento che si segue con carta e penna, otteniamo la situazione mostrata in Figura 14:

Figura 14

```
1010 x 1101 =
-------------
     1010
    0000
   1010
  1010
-------------
 10000010
```

Osserviamo che moltiplicando tra loro due numeri a **4** bit, otteniamo un risultato che richiede al massimo **8** bit; la **CPU** tiene conto di questo aspetto, e quindi e' impossibile che la moltiplicazione possa provocare un overflow.
Notiamo anche l'estrema semplicita' del calcolo dei prodotti parziali; considerando il primo fattore **1010b**, si possono presentare solamente i due casi seguenti:

a) 1010b x 0b = 0000b
b) 1010b x 1b = 1010b

La Figura 15 mostra in forma simbolica la moltiplicazione tra i due nibble **A3A2A1A0** e **B3B2B1B0**:

Figura 15

```
          (A3 A2 A1 A0) x (B3 B2 B1 B0) =
-----------------------------------------------
                        A3xB0 A2xB0 A1xB0 A0xB0
                  A3xB1 A2xB1 A1xB1 A0xB1
            A3xB2 A2xB2 A1xB2 A0xB2
      A3xB3 A2xB3 A1xB3 A0xB3
-----------------------------------------------
  P7    P6    P5    P4    P3    P2    P1    P0
```

Cominciamo con l'osservare che le singole cifre dei prodotti parziali non sono altro che moltiplicazioni tra singoli bit; nel precedente capitolo abbiamo visto che il prodotto binario e' ottenibile attraverso una porta **AND** a **2** ingressi. Nel caso di Figura 15 abbiamo ad esempio:

```
A3 x B2 = A3 AND B2
```

Osserviamo ora che la cifra **P0** del prodotto finale (cifra meno significativa) si ottiene direttamente da **A0xB0**; abbiamo quindi:

```
P0 = A0 AND B0
```

La cifra **P1** del prodotto finale e' data da:

```
P1 = (A0 AND B1) + (A1 AND B0)
```

Per svolgere questa somma utilizziamo un **H.A.** che produce in uscita la cifra **P1** e il riporto **C1** destinato alla colonna successiva.
La cifra **P2** del prodotto finale e' data da:

```
P2 = (A0 AND B2) + (A1 AND B1) + (A2 AND B0) + C1
```

Per svolgere la prima somma **(A0 AND B2) + (A1 AND B1)** utilizziamo un **H.A.** che produce in uscita la somma parziale **P2'** e il riporto **C2'** destinato alla colonna successiva; per svolgere la seconda somma tra **P2'** e **(A2 AND B0)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C1** derivante dal calcolo di **P1**, e produce in uscita la cifra **P2** e il riporto **C2''** destinato alla colonna successiva.

La cifra **P3** del prodotto finale e' data da:

```
P3 = (A0 AND B3) + (A1 AND B2) + (A2 AND B1) + (A3 AND B0) + C2' + C2''
```

Per svolgere la prima somma **(A0 AND B3) + (A1 AND B2)** utilizziamo un **H.A.** che produce in uscita la somma parziale **P3'** e il riporto **C3'** destinato alla colonna successiva; per svolgere la seconda somma tra **P3'** e **(A2 AND B1)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C2'** derivante dal calcolo di **P2**, e produce in uscita la somma parziale **P3''** e il riporto **C3''** destinato alla colonna successiva. Per svolgere la terza somma tra **P3''** e **(A3 AND B0)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C2''** derivante dal calcolo di **P2**, e produce in uscita la cifra **P3** e il riporto **C3'''** destinato alla colonna successiva.
La cifra **P4** del prodotto finale e' data da:

```
P4 = (A1 AND B3) + (A2 AND B2) + (A3 AND B1) + C3' + C3'' + C3'''
```

Per svolgere la prima somma **(A1 AND B3) + (A2 AND B2)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C3'** derivante dal calcolo di **P3**, e produce in uscita la somma parziale **P4'** e il riporto **C4'** destinato alla colonna successiva; per svolgere la seconda somma tra **P4'** e **(A3 AND B1)** utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C3''** derivante dal calcolo di **P3**, e produce in uscita la somma parziale **P4''** e il riporto **C4''** destinato alla colonna successiva. A questo punto utilizziamo un **H.A.** per sommare **P4''** con il riporto **C3'''** derivante dal calcolo di **P3**; in questo modo otteniamo in uscita dall'**H.A.** la cifra **P4** e il riporto **C4'''** destinato alla colonna successiva.
La cifra **P5** del prodotto finale e' data da:

```
P5 = (A2 AND B3) + (A3 AND B2) + C4' + C4'' + C4'''
```

Per svolgere questa somma utilizziamo un **F.A.** che riceve in ingresso anche il riporto **C4'** derivante dal calcolo di **P4**, e produce in uscita la somma parziale **P5'** e il riporto **C5'** destinato alla colonna successiva. A questo punto utilizziamo un **F.A.** per sommare **P5'** con i riporti **C4''** e **C4'''** derivanti dal calcolo di **P4**; in questo modo otteniamo in uscita dal **F.A.** la cifra **P5** e il riporto **C5''** destinato alla colonna successiva.
La cifra **P6** del prodotto finale e' data da:

```
P6 = (A3 AND B3) + C5' + C5''
```

Utilizziamo allora un **F.A.** che riceve in ingresso anche i riporti **C5'** e **C5''** derivanti dal calcolo di **P5**; in questo modo otteniamo in uscita dal **F.A.** la cifra **P6** e il riporto **C6** destinato alla colonna successiva.
Naturalmente il riporto **C6** non e' altro che la cifra **P7** del prodotto finale; quest'ultimo passaggio conferma che nella moltiplicazione l'overflow e' impossibile.

Applicando alla lettera i concetti appena esposti, otteniamo la **R.C.** di Figura 16 che permette di moltiplicare numeri interi senza segno a **4** bit:

Figura 16

La **R.C.** appena descritta, pur essendo perfettamente funzionante, ha un valore puramente teorico; nella pratica infatti si utilizzano **R.C.** che pur essendo basate sul circuito di Figura 16, presentano notevoli ottimizzazioni che hanno lo scopo di aumentare la velocita' di calcolo. In ogni caso appare chiaro il fatto che queste ottimizzazioni, per quanto spinte possano essere, danno luogo ad **R.C.** nettamente piu' complesse di quelle utilizzate dalla **CPU** per le addizioni e per le sottrazioni; proprio per questo motivo, la moltiplicazione impone alla **CPU** un tempo di calcolo nettamente superiore a quello necessario per l'esecuzione delle addizioni e delle sottrazioni.

La **R.C.** di Figura 16 puo' essere utilizzata anche per la moltiplicazione tra numeri interi con segno; a tale proposito bisogna prima di tutto cambiare di segno gli eventuali fattori negativi. In teoria questo metodo funziona, ma la **R.C.** che ne deriva tende a diventare molto piu' complessa di quella di Figura 16; proprio per questo motivo, molte **CPU** utilizzano **R.C.** appositamente concepite per operare direttamente sui numeri interi con segno.

6.7 Divisione tra numeri binari

Anche la divisione viene effettuata dalla **CPU** attraverso un metodo basato sullo stesso procedimento che utilizziamo noi esseri umani quando eseguiamo questa operazione con carta e penna; come viene mostrato in seguito, l'algoritmo utilizzato dalla **CPU** comporta un cambiamento di modulo, per cui e' necessario specificare se vogliamo operare nell'insieme dei numeri senza segno o nell'insieme dei numeri con segno.

Dividendo tra loro due numeri binari a **n** bit, otteniamo un quoziente minore o uguale al dividendo, e quindi interamente rappresentabile attraverso un numero binario a **n** bit; il resto inoltre e' minore o uguale al divisore, per cui anch'esso e' interamente rappresentabile attraverso un numero binario a **n** bit.

Partiamo come al solito dai numeri interi senza segno a **4** bit (nibble), e supponiamo di voler calcolare ad esempio:

```
14 / 5 = Q = 2, R = 4
```

Traducendo tutto in binario, e applicando lo stesso procedimento che si segue con carta e penna, otteniamo la situazione mostrata in Figura 17:

Figura 17

```
1110 / 101 =
101    ------
----   10
 100
 000
----
 100
```

Analizzando la Figura 17 possiamo riassumere i passi necessari per il classico algoritmo di calcolo della divisione:

1) A partire da sinistra si seleziona nel dividendo un numero di bit pari al numero di bit del divisore; la sequenza di bit cosi' ottenuta rappresenta il resto parziale della divisione.

2) Se e' possibile sottrarre il divisore dal resto parziale senza chiedere un prestito, si aggiunge un bit di valore **1** alla destra del quoziente parziale; il risultato della sottrazione rappresenta il nuovo resto parziale. Se invece non e' possibile sottrarre il divisore dal resto parziale senza chiedere un prestito, si aggiunge un bit di valore **0** alla destra del quoziente parziale.
3) Se e' possibile, si aggiunge alla destra del resto parziale un nuovo bit del dividendo e si ricomincia dal punto **2**; se invece non ci sono altri bit del dividendo da aggiungere al resto parziale, la divisione e' terminata con quoziente finale pari all'ultimo quoziente parziale, e resto finale pari all'ultimo resto parziale.

Per poter realizzare un circuito logico capace di effettuare l'operazione di divisione, dobbiamo elaborare il precedente algoritmo in modo da renderlo generale e ripetitivo; dobbiamo cioe' fare in modo che la **CPU** calcoli la divisione ripetendo (iterando) uno stesso procedimento per un determinato numero di volte. Per raggiungere questo obiettivo si utilizza un algoritmo perfettamente equivalente a quello appena illustrato; in riferimento come al solito ai numeri interi senza segno a **4** bit, i passi da compiere sono i seguenti:

1) Si aggiungono quattro **0** alla sinistra del dividendo che viene cosi' convertito in un numero a **8** bit; il numero cosi' ottenuto rappresenta il resto parziale della divisione.
2) Si aggiungono quattro **0** alla destra del divisore che viene cosi' convertito in un numero a **8** bit; le cifre del divisore in sostanza vengono shiftate a sinistra di quattro posti.
3) Le cifre del divisore vengono shiftate a destra di una posizione.
4) Si prova a sottrarre il divisore dal resto parziale senza chiedere un prestito, e se cio' e' possibile si aggiunge un bit di valore **1** alla destra del quoziente parziale, mentre il risultato della sottrazione rappresenta il nuovo resto parziale; se invece la sottrazione non e' possibile, si aggiunge un bit di valore **0** alla destra del quoziente parziale.
5) Si salta al punto **3** e si ripete questo procedimento per **4** volte, pari cioe' al numero di bit dei due operandi; l'ultimo quoziente parziale ottenuto rappresenta il quoziente finale della divisione, mentre l'ultimo resto parziale ottenuto rappresenta il resto finale della divisione.

Proviamo ad applicare questo algoritmo alla precedente divisione di **1110b** per **0101b**; il risultato che si ottiene viene mostrato dalla Figura 18:

Figura 18

```
Il dividendo 1110b diventa 00001110b
Il divisore  0101b diventa 01010000b

1a) il divisore diventa 00101000b
1b) 00001110b - 00101000b non e' possibile e quindi Q' = 0b

2a) il divisore diventa 00010100b
2b) 00001110b - 000101000b non e' possibile e quindi Q' = 00b

3a) il divisore diventa 00001010b
3b) 00001110b - 00001010b = 00000100b e quindi Q' = 001b

4a) il divisore diventa 00000101b
4b) 00000100b - 00000101b non e' possibile e quindi Q' = 0010b

Quoziente finale Q = 0010b
Resto finale     R = 0100b
```

In riferimento quindi ai numeri interi senza segno a **4** bit, possiamo dire che l'algoritmo appena illustrato consiste nel ripetere per **4** volte uno stesso procedimento che comprende uno shift di un posto verso destra delle cifre del divisore, e un tentativo di eseguire una sottrazione tra il resto

parziale e lo stesso divisore.
Nel caso della moltiplicazione, abbiamo risolto il problema attraverso una **R.C.** relativamente semplice; nel caso della divisione invece la situazione si presenta piu' complessa. La necessita' di dover stabilire se effettuare o meno la sottrazione, rende necessario un apposito dispositivo capace di prendere una tale decisione; ci serve cioe' un dispositivo derivato dal **F.S.** e dotato di una linea di controllo attraverso la quale si possa abilitare o meno la sottrazione. Questo dispositivo prende il nome di **Conditional Subtractor** (**C.S.**) o sottrattore condizionale; lo schema circuitale del **C.S.** viene mostrato in Figura 19a, mentre lo schema simbolico viene mostrato in Figura 19b:

Figura 19

Come si puo' notare, la struttura del **C.S.** e' del tutto simile alla struttura del **F.S.**; sono presenti infatti l'ingresso **Ai** per il bit del minuendo, l'ingresso **Bi** per il bit del sottraendo, l'ingresso **Cin** (Carry In) per il prestito richiesto dalla colonna precedente, l'uscita **Cout** (Carry Out) per il prestito richiesto alla colonna successiva e l'uscita **D** per il risultato finale che nel caso del **F.S.** e' dato da:

```
D = Ai - (Bi + Cin)
```

In Figura 19 notiamo pero' anche la presenza della linea **Enable** che attraversa il **C.S.** da **Ein** a **Eout**; proprio attraverso questa linea e' possibile modificare il comportamento del **C.S.**. Come si puo' vedere in Figura 19a, l'ingresso **Ai** e l'uscita **D'** del **F.S.** rappresentano i **2** ingressi di un **MUX** da **2** a **1**; analizzando allora questo circuito possiamo dire che:

a) Se la linea **Enable** e' a livello logico **0**, viene abilitato l'ingresso **D'** del **MUX**, per cui il **C.S.** calcola:

```
D = Ai - (Bi + Cin)
```

b) Se la linea **Enable** e' a livello logico **1**, viene abilitato l'ingresso **Ai** del **MUX**, per cui il **C.S.** calcola:

```
D = Ai
```

Possiamo dire quindi che se la linea **Enable** e' a livello logico **0**, il **C.S.** effettua normalmente la sottrazione; se invece la linea **Enable** e' a livello logico **1**, si ottiene sull'uscita **D** il bit **Ai** del minuendo. Si tenga presente che in Figura 19 la linea **Enable** non ha niente a che vedere con il segnale di abilitazione **E** del **MUX** di Figura 1.
A questo punto si presenta il problema di come pilotare la linea **Enable**; in sostanza, la linea **Enable** deve essere portata a livello logico **0** solo quando il minuendo e' maggiore o uguale al sottraendo. Il metodo che viene utilizzato, consiste nello sfruttare l'uscita **Cout** che viene collegata all'ingresso **Ein**; per capire il funzionamento di questo sistema, analizziamo in dettaglio il funzionamento del **C.S.** di Figura 19:

***** Il **C.S.** effettua la normale sottrazione **D=Ai-(Bi+Cin)**, e ottiene come sappiamo **Cout=0** se non c'e' nessun prestito (**Ai** maggiore o uguale a **Bi**); se invece si verifica un prestito (**Ai** minore di **Bi**), il **C.S.** ottiene **Cout=1**.
***** A questo punto, come e' stato detto in precedenza, l'uscita **Cout** viene collegata a **Ein**, per cui **Ein=Cout=0** conferma il risultato della sottrazione, mentre **Ein=Cout=1** lo annulla e pone **D=Ai**.
In base alle considerazioni appena esposte, si intuisce facilmente che per ottenere la sottrazione condizionale tra due numeri binari a **4** bit, basta collegare in serie **4 C.S.**; il circuito che si ottiene viene mostrato in Figura 20:

Figura 20

Supponiamo ad esempio di avere **A=1110b** e **B=0101b**; in questo caso si vede subito che:

***** Il primo **C.S.** (quello piu' a destra) pone **Cout=1**.
***** Il secondo **C.S.** pone **Cout=0**.
***** Il terzo **C.S.** pone **Cout=0**.
***** Il quarto **C.S.** pone **Cout=0**.

La linea **Enable** si porta quindi a livello logico **0**, e il risultato della sottrazione (**1001b**) viene confermato; se il quarto **C.S.** avesse prodotto **Cout=1**, la linea **Enable** si sarebbe portata a livello logico **1** annullando il risultato della sottrazione e producendo quindi in uscita il nibble **A=1110b**.
In quest'ultimo caso e' come se il minuendo venisse ripristinato (**restored**); proprio per questo motivo, l'algoritmo di calcolo della divisione con i **C.S.** prende il nome di **restoring division** (divisione con ripristino).
Applicando le considerazioni appena esposte, possiamo implementare finalmente il circuito per la divisione tra numeri interi senza segno a **4** bit; ricordando che l'algoritmo esposto in Figura 18 prevede un procedimento ripetitivo costituito da uno shift di un posto verso destra delle cifre del divisore e da un tentativo di sottrazione tra resto parziale e divisore stesso, si ottiene il circuito mostrato in Figura 21:

Figura 21

Gli ingressi **a0**, **a1**, **a2**, **a3**, **a4**, **a5** e **a6** contengono i **4** bit del dividendo e rappresentano anche il resto parziale; naturalmente gli ingressi **a4**, **a5** e **a6** vengono portati a livello logico basso. Gli ingressi **b0**, **b1**, **b2** e **b3** contengono i **4** bit del divisore; come si puo' notare, il divisore si trova shiftato a sinistra di **3** posizioni anziche' di **4** in quanto (Figura 18) **4** shift a sinistra e uno a destra equivalgono a **3** shitf a sinistra.
La riga piu' in alto dei **C.S.** esegue la sottrazione tra **a6a5a4a3** e **b3b2b1b0**; se alla fine non si verifica nessun prestito il risultato della sottrazione viene confermato, mentre in caso contrario viene ripristinato il minuendo, e il risultato prodotto in uscita e' **a6a5a4a3**. Si nota subito che l'uscita **q3** vale **1** se il minuendo e' maggiore o uguale al sottraendo, e **0** in caso contrario; possiamo dire quindi che **q3** non e' altro che uno dei **4** bit del quoziente finale.
Le considerazioni appena esposte si applicano naturalmente anche alle altre tre righe dei **C.S.**; terminata la divisione, sulle quattro uscite **q0**, **q1**, **q2** e **q3** preleviamo i **4** bit del quoziente finale, mentre sulle quattro uscite **r0**, **r1**, **r2** e **r3** preleviamo i **4** bit del resto finale.

Come al solito, il circuito di Figura 21 pur essendo perfettamente funzionante ha un valore puramente teorico; nella pratica infatti si utilizza una versione elaborata e notevolmente ottimizzata del circuito appena descritto. Come e' facile intuire pero', la divisione si presenta come una operazione particolarmente complessa per la **CPU**; infatti la divisione comporta per la **CPU** un tempo di calcolo superiore persino a quello necessario per la moltiplicazione (soprattutto nel caso di restoring delle varie sottrazioni).

Il circuito di Figura 21 puo' essere utilizzato anche per la divisione tra numeri interi con segno; a tale proposito bisogna prima di tutto cambiare di segno gli eventuali operandi negativi. In relazione alla divisione la **CPU** ha bisogno di distinguere tra numeri con o senza segno in quanto abbiamo visto che per poter utilizzare il circuito di Figura 21, e' necessario modificare l'ampiezza in bit del dividendo e del divisore; tutto cio' dimostra che il metodo di calcolo della divisione utilizzato dalla **CPU**, provoca un cambiamento di modulo.

6.8 Scorrimento dei bit di un numero binario

Le considerazioni esposte in precedenza ci hanno permesso di constatare che la moltiplicazione e la divisione sono operazioni particolarmente impegnative per la **CPU**, e richiedono tempi di calcolo notevolmente superiori a quelli necessari per l'addizione o per la sottrazione; purtroppo nel caso generale non e' possibile ovviare a questo problema. Nei precedenti capitoli pero' abbiamo visto che per la moltiplicazione e la divisione esiste un caso particolare che ci permette di velocizzare notevolmente queste operazioni; esaminiamo allora questo caso particolare che e' legato ancora una volta alla struttura dei sistemi di numerazione posizionali.

6.8.1 Moltiplicazione di un numero intero binario per una potenza intera del 2

Nel caso ad esempio della base **10**, abbiamo visto che moltiplicare un numero intero **n** per **10** (cioe' per la base), equivale ad aggiungere uno zero alla destra del numero stesso. Tutte le cifre di **n** scorrono verso sinistra di una posizione aumentando quindi il loro peso; infatti la moltiplicazione di **n** per **10** trasforma le unita' di **n** in decine, le decine in centinaia, le centinaia in migliaia e cosi' via. Analogamente, moltiplicare **n** per **100** (10^2) equivale ad aggiungere due zeri alla destra di **n** facendo scorrere tutte le sue cifre di due posizioni verso sinistra; moltiplicare in generale **n** per 10^m equivale ad aggiungere **m** zeri alla destra di **n** facendo scorrere tutte le sue cifre di **m** posizioni verso sinistra.
Nel caso ancora piu' generale di un numero intero **n** espresso in base **b** qualunque, moltiplicare **n** per b^m equivale ad aggiungere **m** zeri alla destra di **n**, facendo scorrere tutte le sue cifre di **m** posizioni verso sinistra. Possiamo dire allora che ogni volta che uno dei due fattori puo' essere espresso sotto forma di potenza intera della base, la moltiplicazione si traduce in uno scorrimento

verso sinistra delle cifre dell'altro fattore, di un numero di posizioni pari all'esponente della base; risulta evidente che in questo caso, i calcoli diventano estremamente semplici e veloci rispetto al metodo tradizionale.
Proprio per questo motivo, abbiamo visto che le **CPU** della famiglia **80x86** forniscono l'istruzione **SHL** (**Shift Logical Left**) per gli scorrimenti verso sinistra dei bit di un numero intero senza segno, e l'istruzione **SAL** (**Shift Arithmetic Left**) per gli scorrimenti verso sinistra dei bit di un numero intero con segno; siccome pero' il segno viene codificato nel **MSB** di un numero binario, le due istruzioni **SHL** e **SAL** sono assolutamente equivalenti (e quindi interscambiabili).

Per chiarire questi concetti molto importanti, e' necessario tenere presente che le considerazioni appena esposte hanno un valore puramente teorico; in teoria infatti possiamo prendere un numero **n** e possiamo far scorrere i suoi bit di migliaia di posizioni verso sinistra o verso destra. Nel caso del computer pero' il discorso cambia radicalmente; non bisogna dimenticare infatti che la **CPU** gestisce i numeri binari assegnando a ciascuno di essi una ampiezza fissa in bit. Ogni numero binario inoltre viene sistemato in una locazione ad esso riservata; consideriamo ad esempio la situazione mostrata in Figura 22:

Figura 22
0 0 1 0 0 1 1 1

Questa figura mostra una locazione da **8** bit riservata al numero binario **n=00100111b**; il numero binario **n** si trova confinato all'interno della sua locazione da **8** bit, e tutte le modifiche compiute su di esso, devono produrre un risultato interamente rappresentabile attraverso questi **8** bit.
Analizzando allora la Figura 22 si intuisce subito che eccedendo nello scorrimento verso sinistra dei bit di **n**, si rischia di ottenere un risultato sbagliato; il problema che si verifica e' dovuto chiaramente al fatto che un numero eccessivo di scorrimenti puo' provocare la fuoriuscita (dalla parte sinistra della locazione di Figura 22) di cifre significative per il risultato finale.
Supponiamo ad esempio di far scorrere (con **SHL** o con **SAL**) ripetutamente di un posto verso sinistra i bit del numero binario di Figura 22; in questo modo otteniamo la seguente successione:
`01001110b, 10011100b, 00111000b, 01110000b, ...`
Tutti gli scorrimenti si applicano agli **8** bit della locazione di Figura 22, e devono quindi produrre un risultato interamente contenibile nella locazione stessa; e' chiaro quindi che dopo un certo numero di scorrimenti, comincera' a verificarsi il trabocco da sinistra di cifre significative del risultato.
Nell'insieme dei numeri senza segno il numero binario **00100111b** equivale a **39**; il primo scorrimento produce il risultato **78** che e' corretto in quanto rappresenta il prodotto della moltiplicazione di **39** per **2**. Il secondo scorrimento produce il risultato **156** che e' corretto in quanto rappresenta il prodotto della moltiplicazione di **39** per $\mathbf{2^2}$; il terzo scorrimento produce il risultato **56** che e' chiaramente sbagliato. L'errore e' dovuto al fatto che moltiplicando **39** per $\mathbf{2^3}$ si ottiene un numero a **9** bit che non e' piu' contenibile nella locazione di Figura 22; il bit piu' significativo del risultato (di valore **1**) trabocca da sinistra e invalida il risultato finale.

Anche nell'insieme dei numeri con segno il numero binario **00100111b** equivale a **+39**; il primo scorrimento produce il risultato **+78** che e' corretto in quanto rappresenta il prodotto della moltiplicazione di **+39** per **2**. Il secondo scorrimento produce il risultato **-100** che e' chiaramente sbagliato; l'errore questa volta e' dovuto al fatto che moltiplicando **+39** per $\mathbf{2^2}$ si ottiene un numero positivo (**+156**) piu' grande di **+127** che e' il massimo numero positivo rappresentabile con **8** bit.
Osserviamo infatti che il bit di segno che inizialmente valeva **0**, e' traboccato da sinistra, e il suo posto e' stato preso da un bit di valore **1**; il numero **n** quindi si e' trasformato da positivo in negativo.

Le considerazioni appena esposte ci fanno capire che se intendiamo utilizzare **SHL** o **SAL** per calcolare rapidamente il prodotto di un numero binario per una potenza intera del **2**, dobbiamo essere certi della validita' del risultato che verra' fornito da queste istruzioni; in sostanza, dobbiamo verificare che nella parte sinistra del numero **n** ci sia un certo "spazio di sicurezza". Nel caso dei numeri senza segno dobbiamo verificare che nella parte sinistra di **n** ci sia un numero sufficiente di zeri; nel caso dei numeri con segno dobbiamo verificare che nella parte sinistra di **n** ci sia un numero sufficiente di bit di segno (o tutti **0** o tutti **1**).

6.8.2 Divisione di un numero intero binario per una potenza intera del 2

Passando alla divisione sappiamo che in base **10**, dividendo un numero intero positivo **n** per **10**, si provoca uno scorrimento di una posizione verso destra delle cifre di **n**; la cifra meno significativa di **n** (cioe' quella piu' a destra) in seguito allo scorrimento, "trabocca" da destra e viene persa. Tutte le cifre di **n** scorrono verso destra di una posizione riducendo quindi il loro peso; infatti la divisione di **n** per **10** trasforma le decine di **n** in unita', le centinaia in decine, le migliaia in centinaia e cosi' via.
La cifra che trabocca da destra rappresenta il resto della divisione; abbiamo ad esempio:
`138 / 10 = Q = 13, R = 8`
Quindi le tre cifre di **138** scorrono verso destra di una posizione facendo uscire l'**8** che rappresenta proprio il resto della divisione.
Analogamente:
`138 / 100 = Q = 1, R = 38`
Le tre cifre di **138** quindi scorrono verso destra di due posizioni facendo uscire il **38** che anche in questo caso, rappresenta il resto della divisione.
Generalizzando, nel caso di una base **b** qualunque, dividendo un numero intero positivo **n** per b^m si provoca uno scorrimento di **m** posizioni verso destra delle cifre di **n** facendo cosi' traboccare le **m** cifre piu' a destra; le cifre rimaste rappresentano il quoziente, mentre le **m** cifre traboccate da destra rappresentano il resto.
In definitiva, se il divisore puo' essere espresso sotto forma di potenza intera della base, utilizzando questa tecnica si esegue la divisione in modo nettamente piu' rapido rispetto al metodo tradizionale; proprio per questo motivo, abbiamo gia' visto che le **CPU** della famiglia **80x86** forniscono l'istruzione **SHR** (**Shift Logical Right**) per gli scorrimenti verso destra dei bit di un numero intero senza segno, e l'istruzione **SAR** (**Shift Arithmetic Right**) per gli scorrimenti verso destra dei bit di un numero intero con segno. Questa volta pero' le due istruzioni **SHR** e **SAR** non sono equivalenti; la differenza di comportamento tra **SHR** e **SAR** e' legata al fatto che nello scorrimento verso destra dei bit di un numero, e' necessario distinguere tra numeri con segno e numeri senza segno.
Osserviamo infatti che:
`+138 / 10 = Q = +13, R = +8`, mentre `-138 / 10 = Q = -13, R = -8`
Vediamo allora come viene gestita questa situazione da **SHR** e da **SAR**; a tale proposito consideriamo la situazione di Figura 23 che rappresenta come al solito un numero binario contenuto in una locazione da **8** bit:

Figura 23
1 0 1 1 0 0 0 0

Nell'insieme dei numeri interi senza segno, il numero binario **10110000b** equivale a **176**; utilizziamo allora **SHR** per far scorrere ripetutamente di un posto verso destra i bit del numero binario di Figura 23. In questo modo otteniamo la seguente successione:
`01011000b, 00101100b, 00010110b, 00001011b, 00000101b, ...`
Il primo scorrimento produce il risultato **88** (con resto **0b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per **2**; il secondo scorrimento produce il risultato **44** (con resto **00b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per 2^2. Il terzo scorrimento produce il risultato **22** (con resto **000b**) che e' corretto in quanto rappresenta il quoziente della

divisione di **176** per 2^3; il quarto scorrimento produce il risultato **11** (con resto **0000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per 2^4. Il quinto scorrimento produce il risultato **5** (con resto **10000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **176** per 2^5.
Nel caso generale possiamo dire allora che l'utilizzo di **SHR** per la divisione rapida di un numero intero senza segno per una potenza intera del **2**, produce sempre il risultato corretto; la sequenza di cifre che trabocca da destra rappresenta sempre il resto della divisione appena effettuata.
Osserviamo poi che **SHR** opera sui numeri interi positivi, per cui riempie con degli zeri i posti che si liberano a sinistra in seguito allo scorrimento verso destra dei bit di un numero binario; in questo modo si ottiene sempre un quoziente positivo. E' chiaro allora che dopo un certo numero di scorrimenti verso destra dei bit di un numero positivo **n**, il quoziente diventa **0**; nel caso ad esempio di un numero binario a **8** bit, dopo **8** scorrimenti verso destra delle cifre di questo numero si ottiene sicuramente un quoziente nullo.

Nell'insieme dei numeri interi con segno, il numero binario **10110000b** di Figura 23 equivale a **-80**; utilizziamo allora **SAR** per far scorrere ripetutamente di un posto verso destra i bit di questo numero binario. In questo modo otteniamo la seguente successione:
`11011000b, 11101100b, 11110110b, 11111011b, 11111101b, ...`
Osserviamo subito che **SAR** riempie con degli **1** i posti che si liberano a sinistra del numero **10110000b** in seguito allo scorrimento delle sue cifre verso destra; questo perche' il **MSB** di **10110000b** e' **1**, e quindi **SAR** deve preservare il bit di segno.
Il primo scorrimento produce il risultato **-40** (con resto **0b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per **2**; il secondo scorrimento produce il risultato **-20** (con resto **00b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per 2^2. Il terzo scorrimento produce il risultato **-10** (con resto **000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per 2^3; il quarto scorrimento produce il risultato **-5** (con resto **0000b**) che e' corretto in quanto rappresenta il quoziente della divisione di **-80** per 2^4. Il quinto scorrimento produce il risultato **-3** (con resto **10000b**) che e' pero' sbagliato in quanto dividendo **-80** per 2^5 si dovrebbe ottenere il quoziente **-2** e il resto **-16**!
Dall'analisi dei risultati ottenuti con **SHR** e con **SAR** emerge allora che:

***** Nel caso dei numeri interi senza segno, **SHR** produce sempre il quoziente e il resto corretti.
***** Nel caso dei numeri interi con segno `positivi`, **SAR** produce sempre il quoziente e il resto corretti.
***** Nel caso dei numeri interi con segno `negativi`, **SAR** produce il quoziente e il resto corretti solamente quando il resto e' zero; se invece il resto e' diverso da zero, il quoziente ottenuto appare sommato a **-1**.

Per capire il perche' di questa situazione, bisogna tenere presente che la **CPU**, nell'eseguire una divisione intera tra due numeri interi con o senza segno, arrotonda sempre il quoziente verso lo zero; questo comportamento e' corretto in quanto rispetta le regole matematiche della divisione intera. Ad esempio, il quoziente **+4.9** deve essere arrotondato a **+4** e non a **+5**; analogamente, il quoziente **-6.8** deve essere arrotondato a **-6** e non a **-7**!

Con l'istruzione **SHR**, questa regola matematica viene sempre rispettata; cio' e' dovuto al metodo utilizzato dalla **CPU**, per codificare i numeri interi senza segno.
In riferimento, ad esempio, ai numeri interi senza segno a **8** bit, sappiamo che la **CPU** utilizza le codifiche binarie comprese tra **00000000b** e **11111111b**, per rappresentare i numeri interi senza segno compresi, rispettivamente, tra **0** e **255**; come si puo' notare, i numeri interi senza segno crescenti (**0**, **1**, **2**, etc) hanno una codifica binaria crescente (**00000000b**, **00000001b**, **00000010b**, etc).

La conseguenza pratica di tutto cio' e' data dal fatto che, utilizzando l'istruzione **SHR** per far scorrere di **n** posti verso destra le cifre di un numero intero senza segno **m**, si ottiene sempre un risultato arrotondato verso lo zero; tale risultato, coincide quindi con il quoziente della divisione intera tra **m** e 2^n, e inoltre, le **n** cifre traboccate da destra, coincidono con il resto della divisione stessa!

Con l'istruzione **SAR**, questa regola matematica non viene sempre rispettata; cio' e' dovuto al metodo utilizzato dalla **CPU**, per codificare i numeri interi con segno (codifica binaria in complemento a **2**).
In riferimento, ad esempio, ai numeri interi con segno a **8** bit, sappiamo che:

***** le codifiche binarie comprese tra **00000000b** e **01111111b**, rappresentano i numeri interi positivi compresi, rispettivamente, tra **0** e **+127**;
***** le codifiche binarie comprese tra **10000000b** e **11111111b**, rappresentano i numeri interi negativi compresi, rispettivamente, tra **-128** e **-1**.

Come si puo' notare, i numeri interi positivi crescenti (**+0**, **+1**, **+2**, etc), hanno una codifica binaria crescente (**00000000b**, **00000001b**, **00000010b**, etc); invece, i numeri interi negativi crescenti in valore assoluto (**|-1|**, **|-2|**, **|-3|**, etc), hanno una codifica binaria decrescente (**11111111b**, **11111110b**, **11111101b**, etc)!
La conseguenza pratica di tutto cio' e' data dal fatto che, utilizzando l'istruzione **SAR** per far scorrere di **n** posti verso destra le cifre di un numero intero positivo **m**, si ottiene sempre un risultato arrotondato verso lo zero; tale risultato, coincide quindi con il quoziente della divisione intera tra **m** e 2^n, e inoltre, le **n** cifre traboccate da destra, coincidono con il resto della divisione stessa.
Utilizzando l'istruzione **SAR** per far scorrere di **n** posti verso destra le cifre di un numero intero negativo **m**, si ottiene sempre un risultato arrotondato verso l'**infinito negativo**; ad esempio, il risultato **-5.3** viene arrotondato a **-6** anziche' a **-5**.
Se **m** e' un numero intero negativo multiplo intero di 2^n, si ottiene un risultato che coincide con il quoziente della divisione intera tra **m** e 2^n; infatti, in questo caso il resto e' zero, e quindi il quoziente non necessita di nessun arrotondamento (quoziente esatto).
Se **m** e' un numero intero negativo non divisibile per 2^n, si ottiene un risultato che non coincide con il quoziente della divisione intera tra **m** e 2^n; infatti, in questo caso il resto e' diverso da zero, e quindi il quoziente necessita di un arrotondamento. Mentre pero' la divisione arrotonda il suo quoziente verso lo zero, l'istruzione **SAR** arrotonda il suo risultato verso l'infinito negativo (ad esempio, il valore **-9.3** viene arrotondato a **-9** dalla divisione, e a **-10** da **SAR**); in tal caso, il risultato prodotto da **SAR** differisce del valore **-1** rispetto al quoziente della divisione!

I problemi appena illustrati, inducono decisamente ad evitare l'uso degli scorrimenti aritmetici a destra per dividere rapidamente numeri negativi per potenze intere del **2**; appare molto piu' sensato, in questo caso, ricorrere alla divisione tradizionale che comporta pero', come sappiamo, una notevole lentezza di esecuzione.

6.8.3 Verifica del risultato attraverso i flags

Come accade per tutte le altre operazioni logico aritmetiche, la **CPU** utilizza i vari flags per fornire al programmatore tutti i dettagli sul risultato di una operazione appena eseguita; analizziamo innanzi tutto le informazioni che ci vengono fornite attraverso **CF**, **SF** e **OF** in relazione al risultato prodotto dalle istruzioni **SHL**, **SAL**, **SHR** e **SAR** applicate ad un numero binario **n**.
Il flag **CF** contiene sempre l'ultimo bit traboccato da **n** (da destra o da sinistra); se ad esempio facciamo scorrere di un posto verso destra i bit di **01001101b**, si ottiene **CF=1** in quanto il bit traboccato da destra ha valore **1**.

Il flag **SF** contiene sempre il **MSB** del risultato appena ottenuto; se ad esempio facciamo scorrere di un posto verso sinistra i bit di **01001101b**, si ottiene il risultato **10011010b**, e quindi **SF=1** in quanto il **MSB** ha assunto il valore **1**.
Il flag **OF** si porta a **1** ogni volta che il **MSB** del risultato appena ottenuto cambia da **0** a **1** o da **1** a **0**; se ad esempio facciamo scorrere di un posto verso sinistra i bit di **01001101b**, si ottiene il risultato **10011010b**, e quindi **OF=1** in quanto il valore del **MSB** e' cambiato da **0** a **1**.

6.8.4 Esempio di shifter a 4 bit

Vediamo ora come puo' essere implementata una **R.C.** capace di effettuare lo scorrimento dei bit di un numero binario; la Figura 24 illustra uno "**shifter**" a **4** bit capace di implementare tutte le **4** istruzioni **SHL**, **SAL**, **SHR** e **SAR**:

Figura 24

Indichiamo con **A0**, **A1**, **A2** e **A3** i **4** bit del nibble in ingresso, e con **Y0**, **Y1**, **Y2** e **Y3** i **4** bit del nibble in uscita; abbiamo poi l'uscita **CF** che consente di inviare al **Carry Flag** il bit appena scartato, e infine i seguenti tre ingressi di abilitazione:

SHL che abilita lo scorrimento logico a sinistra (equivalente a **SAL**);
SHR che abilita lo scorrimento logico a destra;
SAR che in congiunzione con **SHR** abilita lo scorrimento aritmetico a destra.

Come si puo' constatare attraverso la Figura 23, si ha inoltre:
`SF = Y3` e `OF = A3 EX-NOR Y3`

Ponendo in Figura 23 **SHL=1**, **SHR=0** e **SAR=0**, si abilita la **R.C.** allo scorrimento logico/aritmetico verso sinistra dei bit del nibble in ingresso; come gia' sappiamo, questa operazione produce effetti identici sia sui numeri senza segno, sia sui numeri con segno (**SHL=SAL**).

Ponendo in Figura 23 **SHL=0**, **SHR=1** e **SAR=0**, si abilita la **R.C.** allo scorrimento logico verso destra dei bit del nibble in ingresso; il nibble in ingresso deve essere quindi un numero senza segno.
Ponendo in Figura 23 **SHL=0**, **SHR=1** e **SAR=1**, si abilita la **R.C.** allo scorrimento aritmetico verso destra dei bit del nibble in ingresso; il nibble in ingresso deve essere quindi un numero con segno.

In relazione agli scorrimenti dei bit di un numero binario, e' necessario sottolineare un aspetto molto importante; come si puo' facilmente intuire, piu' aumenta il numero di shift da effettuare, piu' aumentano i tempi per l'esecuzione di questa operazione da parte della **CPU**. E' chiaro allora che al crescere del numero di shift da effettuare diminuiscono i vantaggi di questa tecnica rispetto alle moltiplicazioni e divisioni tradizionali.

6.9 Rotazione dei bit di un numero binario

In conclusione di questo capitolo, citiamo anche l'operazione di rotazione dei bit di un numero binario; questa operazione e' simile a quella di scorrimento dei bit, con la differenza pero' che i bit che traboccano da una estremita' dell'operando, rientrano in sequenza dall'altra estremita' dell'operando stesso. Appare evidente che in relazione alla operazione di rotazione dei bit, non ha nessun senso la distinzione tra numeri senza segno e numeri con segno.

Le **CPU** della famiglia **80x86** mettono a disposizione **4** istruzioni per la rotazione dei bit, che sono **ROL**, **RCL**, **ROR**, **RCR**.
L'istruzione **ROL** (**Rotate Left**) fa' scorrere verso sinistra i bit di un numero binario; i bit che traboccano da sinistra rientrano in sequenza da destra. In Figura 23 si puo' simulare l'istruzione **ROL** abilitando l'ingresso **SHL**, e collegando l'uscita **Y0** all'uscita **CF**.
L'istruzione **RCL** (**Rotate Through Carry Left**) fa' scorrere verso sinistra i bit di un numero binario; i bit che traboccano da sinistra finiscono in sequenza nel flag **CF**, mentre i vecchi bit contenuti in **CF** rientrano in sequenza dalla destra del numero binario. In Figura 23 si puo' simulare l'istruzione **RCL** abilitando l'ingresso **SHL** e memorizzando in **Y0** il vecchio contenuto di **CF** (prima di ogni shitf).
L'istruzione **ROR** (**Rotate Right**) fa' scorrere verso destra i bit di un numero binario; i bit che traboccano da destra rientrano in sequenza da sinistra. In Figura 23 si puo' simulare l'istruzione **ROR** abilitando l'ingresso **SHR**, e collegando l'uscita **Y3** all'uscita **CF**.
L'istruzione **RCR** (**Rotate Through Carry Right**) fa' scorrere verso destra i bit di un numero binario; i bit che traboccano da destra finiscono in sequenza nel flag **CF**, mentre i vecchi bit contenuti in **CF** rientrano in sequenza dalla sinistra del numero binario. In Figura 23 si puo' simulare l'istruzione **RCR** abilitando l'ingresso **SHR** e memorizzando in **Y3** il vecchio contenuto di **CF** (prima di ogni shitf).

Dalle considerazioni appena esposte si intuisce che la **R.C.** di Figura 23 puo' essere facilmente modificata per poter implementare sia le istruzioni di scorrimento, sia le istruzioni di rotazione dei bit; nella pratica si segue proprio questa strada che permette di ottenere un notevole risparmio di porte logiche.

Capitolo 7 - L'architettura generale del computer

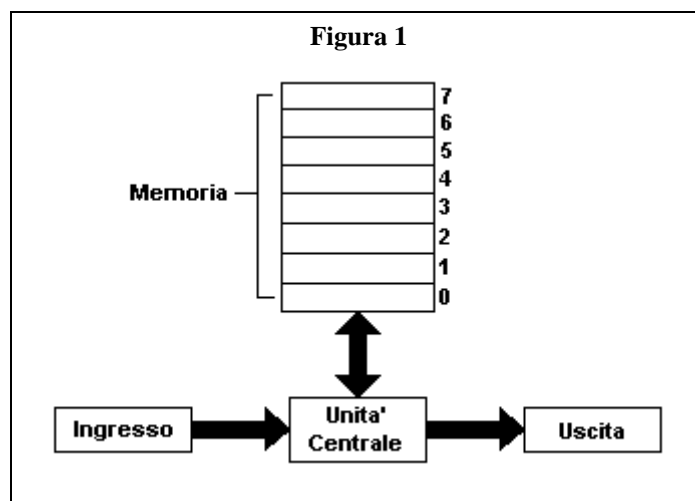
Come e' stato detto in un precedente capitolo, sin dall'antichita' gli esseri umani hanno cercato di realizzare macchine capaci di svolgere in modo automatico calcoli piu' o meno complessi; un esempio famoso e' rappresentato dalla calcolatrice meccanica di **Pascal** (**1642**) che attraverso un sistema di ingranaggi era in grado di eseguire semplici addizioni. Un altro esempio famoso e' rappresentato dalle macchine calcolatrici realizzate dal matematico **Charles Babbage** a partire dal **1822**; queste macchine formate da ingranaggi, cinghie, pulegge, catene di trasmissione, etc, divennero via via sempre piu' sofisticate sino a permettere il calcolo di equazioni differenziali. Il caso delle macchine di **Babbage** assume una importanza enorme nella storia del computer in quanto questo scienziato aveva capito che se una macchina e' in grado di svolgere un determinato calcolo, allora puo' essere progettata in modo da svolgere qualsiasi altro calcolo; il ragionamento di **Babbage** venne dimostrato matematicamente parecchi anni dopo da **Alan Turing**. Si puo' dire che **Babbage** aveva intuito con piu' di un secolo di anticipo l'idea fondamentale su cui si basano gli odierni calcolatori elettronici; questo scienziato non riusci' pero' a mettere in pratica le sue teorie in quanto la tecnologia disponibile nel **XVIII** secolo per la realizzazione di queste macchine era esclusivamente di tipo meccanico.

Le considerazioni appena esposte ci fanno capire che le prime macchine di calcolo automatico presentavano una architettura piuttosto rigida che consentiva loro di eseguire un numero limitato di operazioni; le istruzioni di elaborazione infatti venivano incorporate (cablate) direttamente nei meccanismi della macchina stessa e ogni volta che si presentava la necessita' di eseguire un calcolo differente, bisognava progettare una nuova macchina o, nella migliore delle ipotesi, bisognava modificarne la struttura interna.
Le macchine calcolatrici di questo tipo vengono definite a **logica cablata** e presentano come principale difetto la scarsissima flessibilita'; il pregio fondamentale di una macchina a logica cablata e' rappresentato invece dalla elevatissima velocita' di calcolo.

7.1 La macchina di Von Neumann

All'inizio del **XIX** secolo, nel mondo delle macchine di calcolo automatico e' scoppiata una autentica rivoluzione legata principalmente al nome dello scienziato **John Von Neumann**; come e' stato detto nel Capitolo 2, **Von Neumann** ha osservato che nella risoluzione di un problema attraverso una macchina, si possono individuare tre fasi principali:

a) immissione dei dati in ingresso;
b) elaborazione dei dati appena immessi;
c) presentazione in uscita dei risultati.

Queste considerazioni hanno portato **Von Neumann** alla conclusione che i calcolatori dovessero avere una struttura come quella schematizzata in Figura 1:

In questa struttura chiamata **Macchina di Von Neumann** notiamo la presenza di un dispositivo di **Ingresso** o **Input** attraverso il quale vengono inseriti i dati da elaborare; il dispositivo chiamato **Memoria** puo' essere visto come una sorta di "magazzino" all'interno del quale vengono sistemati temporaneamente i dati appena immessi. L'**Unita' Centrale di Elaborazione** o **Central Processing Unit** (**CPU**) e' il cuore di tutto il sistema ed ha il compito di elaborare i dati immagazzinati nella memoria; infine il dispositivo di **Uscita** o **Output** ha il compito di presentare i risultati delle elaborazioni appena effettuate.
Gli studi di **Von Neumann** hanno consentito la realizzazione di macchine di calcolo automatico caratterizzate da semplicita' costruttiva, potenza e flessibilita' al punto che ancora oggi, moltissime famiglie di computers (piattaforme hardware) si ispirano proprio a questo tipo di architettura; in particolare possiamo citare i **PC** basati sulla famiglia dei processori **Intel 80x86** e compatibili.

7.2 Il programma

L'aspetto veramente rivoluzionario che caratterizza una macchina di **Von Neumann** e' dato dal fatto che la memoria e' destinata a contenere non solo i dati da elaborare, ma anche le relative istruzioni di elaborazione; nel loro insieme i dati da elaborare e le istruzioni di elaborazione formano un cosiddetto **programma**.

Si puo' immaginare la memoria come un grande contenitore suddiviso in tanti scomparti ciascuno dei quali rappresenta una cosiddetta **cella di memoria**; per individuare in modo univoco le celle, assegnamo a ciascuna di esse un numero progressivo **0**, **1**, **2**, **3**, etc, chiamato **indirizzo di memoria**. Grazie agli indirizzi di memoria, possiamo accedere ad ogni cella sia per immagazzinare in essa un dato (**accesso in scrittura**), sia per leggere il suo contenuto (**accesso in lettura**).

Vediamo ora come avviene l'esecuzione di un programma in una macchina come quella di Figura 1. Riserviamo un'area della memoria ai dati in ingresso e a quelli in uscita; quest'area rappresenta il **blocco dati** del programma. Riserviamo un'altra area della memoria alle istruzioni di elaborazione; quest'altra area rappresenta il **blocco codice** del programma. Inizializziamo poi un apposito **contatore** caricando in esso l'indirizzo di memoria dove si trova la prima istruzione da eseguire; a questo punto il controllo passa alla **CPU** che puo' iniziare la fase di elaborazione del programma.

Supponiamo ad esempio di avere in memoria il programma mostrato in Figura 2, con il contatore che inizialmente contiene l'indirizzo **0000**:

Figura 2

Indirizzo	**Contenuto**
0000	Leggi il dato contenuto in 0005
0001	Leggi il dato contenuto in 0006
0002	Somma i due dati
0003	Scrivi il risultato in 0007
0004	Fine programma
0005	3500
0006	2300
0007	----

La **CPU** va' all'indirizzo **0000** dove trova una istruzione che consiste nella lettura del dato (**3500**) che si trova in memoria all'indirizzo **0005**; mentre la **CPU** esegue questa istruzione, il contatore viene incrementato di **1** e il suo contenuto diventa quindi **0001**.
La **CPU** va' all'indirizzo **0001** dove trova una istruzione che consiste nella lettura del dato (**2300**) che si trova in memoria all'indirizzo **0006**; mentre la **CPU** esegue questa istruzione, il contatore viene incrementato di **1** e il suo contenuto diventa quindi **0002**.
La **CPU** va' all'indirizzo **0002** dove trova una istruzione che consiste nel calcolo della somma dei due dati appena letti dalla memoria; mentre la **CPU** esegue questa istruzione, il contatore viene incrementato di **1** e il suo contenuto diventa quindi **0003**.
La **CPU** va' all'indirizzo **0003** dove trova una istruzione che consiste nel salvare il risultato della somma (**5800**) nella cella di memoria **0007**; come si puo' notare, la cella **0007** e' inizialmente vuota ed e' destinata a contenere il risultato delle elaborazioni effettuate dalla **CPU**. Mentre la **CPU** esegue questa istruzione, il contatore viene incrementato di **1** e il suo contenuto diventa quindi **0004**.
La **CPU** va' all'indirizzo **0004** dove trova una istruzione che segnala la fine delle elaborazioni; a questo punto la **CPU** entra in pausa in attesa di ulteriori richieste di elaborazione.
Non ci vuole molto a capire che il sistema appena descritto presenta una elevatissima flessibilita'; se vogliamo sostituire ad esempio l'addizione con una sottrazione, ci basta modificare le apposite istruzioni del programma appena illustrato. Naturalmente le varie istruzioni devono appartenere ad un insieme riconosciuto dalla **CPU**; questo insieme viene definito **set di istruzioni della CPU**.
Una macchina di calcolo automatico come quella appena descritta viene definita a **logica programmabile**, e presenta l'eccezionale caratteristica di poter svolgere praticamente un numero illimitato di compiti differenti; naturalmente le macchine di questo tipo sono enormemente piu' complesse e quindi piu' lente rispetto alle macchine a logica cablata.

7.3 Il microprocessore

Un'altra rivoluzione che ha letteralmente sconvolto il mondo delle macchine di calcolo automatico e' stata provocata dall'invenzione del **microprocessore** avvenuta nel **1970**; questa invenzione e' legata anche al nome dell'italiano **Federico Faggin** che all'epoca lavorava come progettista in una piccola e giovane societa' della **Silicon Valley** chiamata **Intel**. Insieme ai suoi colleghi **Faggin** si rese conto che l'evoluzione tecnologica aveva raggiunto ormai un livello tale da consentire la possibilita' di miniaturizzare le porte logiche e di inserirle in grande quantita' in uno spazio estremamente ristretto; in particolare, i progettisti della **Intel** riuscirono a fabbricare interi circuiti logici direttamente all'interno di un piccolo blocco di materiale semiconduttore chiamato **chip**. In

questo modo ottennero un microcircuito che integrava al suo interno numerose **R.C.** come quelle presentate nel precedente capitolo; attraverso un apposito sistema di controllo esterno, questo microcircuito che venne chiamato appunto **microprocessore**, poteva essere pilotato in modo da fargli svolgere una numerosa serie di operazioni logico aritmetiche.
Il termine **microprocessore** e' sinonimo quindi di micro **CPU**; nel caso generale comunque il termine **CPU** indica una rete logica complessa capace di eseguire una serie piu' o meno numerosa di operazioni logico aritmetiche.
La prima **CPU** realizzata dal gruppo di progettisti di cui faceva parte anche **Faggin** e' stata chiamata **4004** ed aveva una architettura a **4** bit; nel **1972** e' nata la **CPU 8008** con architettura a **8** bit. Nel **1974 Faggin** prima di uscire dalla **Intel** progetto' la **CPU 8080** che aveva sempre una architettura a **8** bit, ma garantiva prestazioni notevolmente superiori rispetto alla **8008**; successivamente **Faggin** nel **1975** fondo' la societa' **Zilog** e progetto' la fortunatissima **CPU Z80** a **8** bit che ando' ad equipaggiare i famosissimi micro computers **Sinclair ZX Spectrum**.
A partire dagli anni **80'** sono comparse sul mercato **CPU** sempre piu' potenti e sofisticate; in particolare, la famiglia **80x86** si e' evoluta con la nascita di nuove **CPU** a **16** bit e a **32** bit, sino alle attuali **CPU** a **64** bit. La Figura 3 mostra ad esempio un ingrandimento della struttura interna della **CPU Intel Pentium I** a **64** bit; questa **CPU** (che viene considerata ormai vecchia) integra al suo interno circa **3100000** transistor!

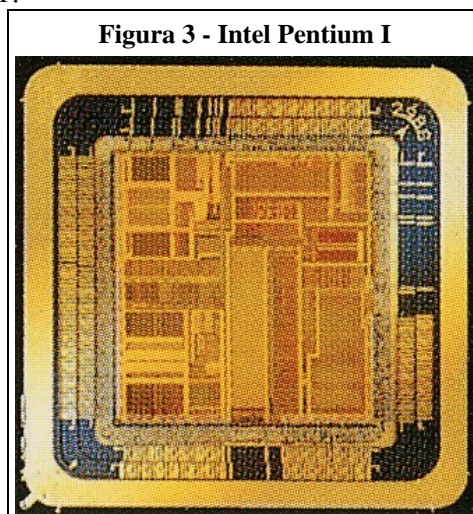
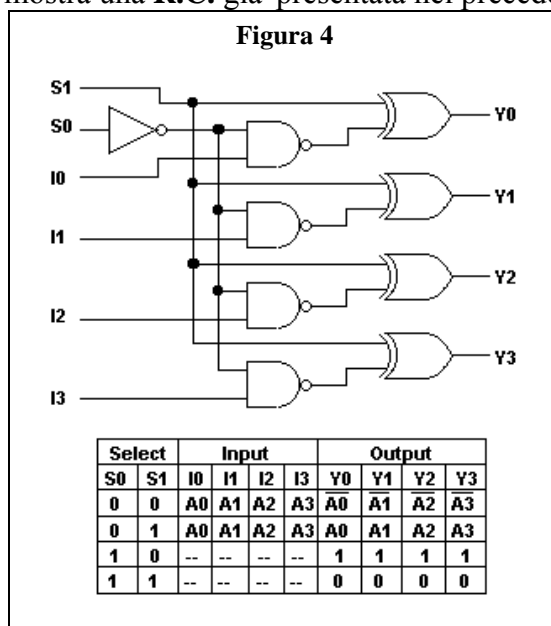
Figura 3 - Intel Pentium I

Cerchiamo ora di capire meglio il principio di funzionamento di una **CPU**; a tale proposito analizziamo la Figura 4 che mostra una **R.C.** gia' presentata nel precedente capitolo:

Figura 4

Select		Input				Output			
S0	S1	I0	I1	I2	I3	Y0	Y1	Y2	Y3
0	0	A0	A1	A2	A3	$\overline{A0}$	$\overline{A1}$	$\overline{A2}$	$\overline{A3}$
0	1	A0	A1	A2	A3	A0	A1	A2	A3
1	0	--	--	--	--	1	1	1	1
1	1	--	--	--	--	0	0	0	0

Come gia' sappiamo, attraverso le due linee di selezione possiamo far svolgere a questo circuito **4** funzioni differenti; in particolare si nota che:

***** Per **S0 = 0** e **S1 = 0** si ottiene in uscita il complemento a **1** del nibble in ingresso.
***** Per **S0 = 0** e **S1 = 1** si ottiene in uscita lo stesso nibble in ingresso.
***** Per **S0 = 1** e **S1 = 0** si ottiene in uscita il nibble **1111b**.
***** Per **S0 = 1** e **S1 = 1** si ottiene in uscita il nibble **0000b**.

Si puo' paragonare il circuito di Figura 4 ad una sorta di rudimentale **CPU** a **4** bit, dotata di **4** linee per il dato in ingresso, **4** linee per il dato in uscita e **2** linee di selezione (linee di controllo) che consentono di specificare una tra le $2^2=4$ istruzioni eseguibili; ciascuna istruzione per essere capita dalla **CPU** deve essere espressa come al solito sotto forma di segnali logici, e cioe' sotto forma di codice binario. Nel caso ad esempio di **S0=0** e **S1=0**, il circuito di Figura 4 esegue l'istruzione **NOT** che consiste in questo caso nella inversione dei **4** bit del dato in ingresso; possiamo dire allora che in questa **CPU** l'istruzione **NOT** viene codificata con il codice binario **00b**. Il valore binario che codifica una determinata istruzione prende il nome di **codice macchina**; l'insieme di tutti i codici macchina riconosciuti da una **CPU** rappresenta come e' stato gia' detto il **set di istruzioni** della **CPU** stessa.
A questo punto appare chiaro il principio di funzionamento di una **CPU**; ciascuna istruzione appartenente al programma da eseguire, viene codificata attraverso il relativo codice macchina.
L'esecuzione di una determinata istruzione consiste nell'invio alla **CPU** della sequenza di segnali logici che codificano l'istruzione stessa; questi segnali selezionano all'interno della **CPU** l'apposita **R.C.** che esegue l'istruzione desiderata.
Tornando al circuito di Figura **4**, supponiamo ad esempio di eseguire un programma che ad un certo punto prevede una istruzione per l'inversione dei bit del nibble che si trova in memoria all'indirizzo **01001111b**; il codice macchina di questa istruzione puo' essere allora:

```
00b 01001111b
```

L'esecuzione di questa istruzione si svolge in diverse fasi; prima di tutto l'istruzione viene decodificata da un apposito circuito di controllo; il circuito di controllo dice alla **CPU** di leggere dalla memoria un nibble che si trova all'indirizzo **01001111b**; una volta che il nibble e' stato caricato (attraverso le **4** linee di ingresso), il circuito di controllo invia il codice **00b** alle linee di selezione della **CPU** in modo da abilitare l'istruzione **NOT**. Alla fine si ottiene in uscita il nibble in ingresso con i bit invertiti.

Le considerazioni appena esposte permettono anche di capire che l'unico linguaggio che viene compreso dalla **CPU** e' naturalmente il codice macchina; nei capitoli successivi vedremo pero' che fortunatamente la **CPU** puo' essere programmata anche attraverso linguaggi molto piu' semplici e allo stesso tempo piu' sofisticati.

7.4 La struttura generale di un computer

In base a cio' che e' stato appena detto a proposito della macchina di **Von Neumann** e del principio di funzionamento di una **CPU**, possiamo subito dedurre una serie di considerazioni generali relative alla organizzazione interna di un computer; l'aspetto piu' evidente riguarda il fatto che la **CPU** ha la necessita' di comunicare con tutti gli altri dispositivi del computer, che vengono chiamati **periferiche**. Tra le varie periferiche, la piu' importante di tutte e' senza dubbio la memoria del computer che viene anche chiamata **memoria centrale**; proprio per questo motivo, quando si parla di **cuore** del computer ci si riferisce all'insieme formato dalla **CPU** e dalla memoria centrale.
Tutte le altre periferiche vengono trattate come secondarie, e vengono anche chiamate **dispositivi di I/O** (input/output); tra i vari dispositivi di **I/O** che usualmente risultano collegati al computer si possono citare: tastiere, mouse, monitor, stampanti, plotter, scanner, schede audio, schede video,

joystick, hard disk, lettori di floppy disk, lettori CD, etc. Esistono dispositivi accessibili sia in lettura che in scrittura (memoria centrale, hard disk, floppy disk), altri accessibili solo in lettura (mouse, joystick) e altri ancora accessibili solo in scrittura (stampanti, plotter).
La **CPU** dialoga con una qualsiasi periferica attraverso un interscambio di dati; questo accade ad esempio quando la **CPU** deve accedere in lettura o in scrittura alla memoria, oppure quando la **CPU** deve leggere le coordinate del cursore del mouse sullo schermo, oppure quando la **CPU** deve visualizzare una immagine sullo schermo, etc. Affinche' sia possibile l'interscambio di dati, la **CPU** viene connessa a tutte le periferiche attraverso una serie di linee elettriche che nel loro insieme formano il cosiddetto **Data Bus** (bus dei dati); su ciascuna linea elettrica del **Data Bus** transita uno dei bit del dato da trasferire.

Un'altro aspetto abbastanza evidente riguarda il fatto che la **CPU** per poter dialogare con una periferica deve chiaramente conoscerne l'indirizzo; nel caso della memoria abbiamo gia' visto che le varie celle vengono identificate attraverso un indirizzo rappresentato da un numero intero. Per tutte le altre periferiche si utilizza lo stesso procedimento; ad ogni periferica viene associato quindi un determinato numero intero che rappresenta l'indirizzo che identifica in modo univoco la periferica stessa.

Possiamo dire quindi che nel momento in cui la **CPU** vuole dialogare con una determinata periferica, deve prima di tutto specificare l'indirizzo a cui vuole accedere; come al solito, questo indirizzo deve essere rappresentato sotto forma di segnali logici, e cioe' sotto forma di codice binario. A tale proposito la **CPU** viene connessa a tutte le periferiche attraverso una serie di linee elettriche che nel loro insieme formano il cosiddetto **Address Bus** (bus degli indirizzi); su ciascuna linea elettrica dell'**Address Bus** transita uno dei bit del numero binario che rappresenta l'indirizzo a cui la **CPU** vuole accedere.

Tutto il sistema appena descritto non potrebbe assolutamente funzionare se non venisse coordinato da un apposito circuito di controllo; questo circuito rappresenta la cosiddetta logica di controllo o **Control Logic** (**CL**) del computer. Il compito fondamentale svolto dalla **CL** consiste nel gestire i vari dispositivi che formano il computer stabilendo istante per istante quali dispositivi devono essere attivati e quali invece devono essere disattivati.

Per avere un'idea dell'importanza della **CL**, supponiamo ad esempio che la **CPU** debba leggere un dato dalla memoria per poi applicare ad esso l'istruzione **NOT**; in questo caso la **CL** carica sull'**Address Bus** l'indirizzo di memoria a cui si deve accedere, e predispone la memoria stessa per una operazione di lettura. Terminata questa fase la **CL** autorizza la **CPU** a leggere il dato dalla memoria attraverso il **Data Bus**; una volta che la lettura e' stata effettuata, la **CL** autorizza l'invio alla **CPU** del codice macchina che abilita l'esecuzione dell'istruzione **NOT** sul dato appena letto.

Per la corretta gestione di tutta questa situazione, la **CPU** viene connessa a tutte le periferiche attraverso una serie di linee elettriche che nel loro insieme formano il cosiddetto **Control Bus** (bus di controllo); sul **Control Bus** transitano dei valori binari che codificano i vari segnali di controllo necessari per il coordinamento di tutto il sistema.

Raccogliendo le considerazioni appena esposte, possiamo definire la struttura generale che assume un classico computer appartenente alla piattaforma hardware basata sulle **CPU Intel 80x86** e compatibili; la Figura 5 illustra uno schema a blocchi che comprende anche diverse periferiche collegate al computer:

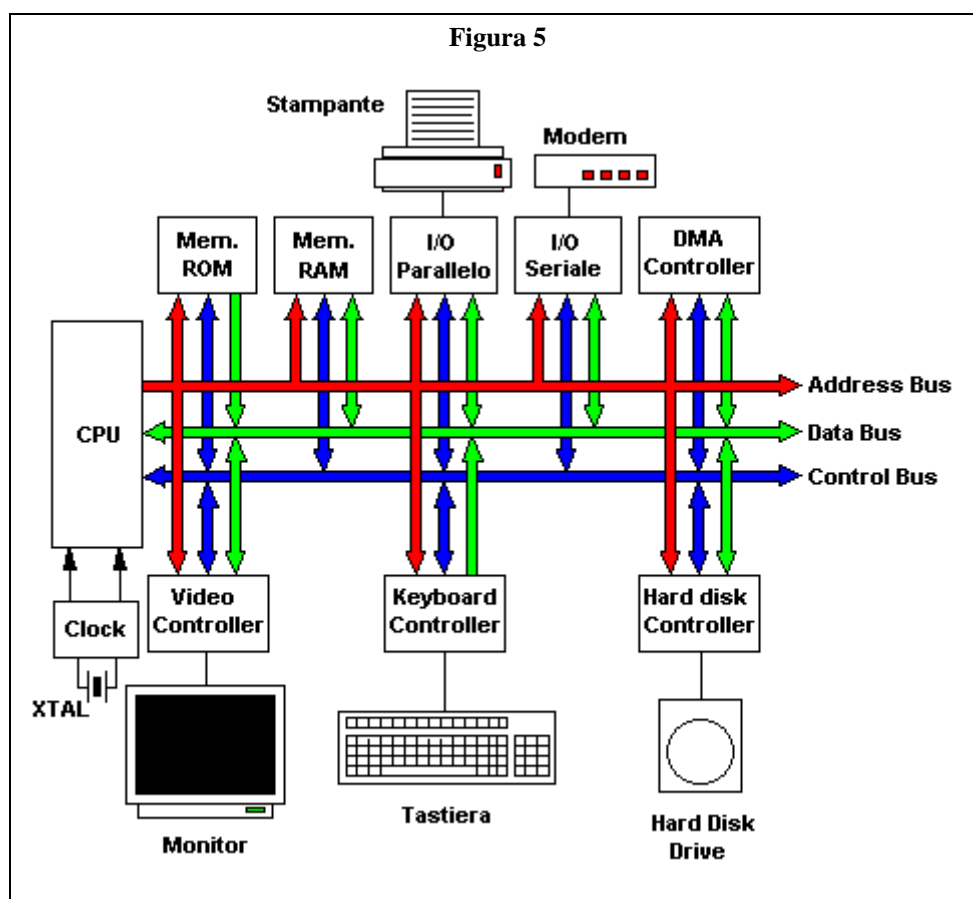



Figura 5

Analizziamo innanzi tutto l'**Address Bus** che in Figura 5 e' stato evidenziato con il colore rosso; l'ampiezza dell'**Address Bus**, e cioe' il numero di linee da cui e' composto, determina la quantita' massima di memoria fisicamente indirizzabile dalla **CPU**. Osserviamo infatti che su un **Address Bus** a **n** linee possono transitare numeri binari formati da **n** bit (un bit per ogni linea); con **n** bit possiamo rappresentare 2^n numeri interi positivi distinti, e cioe' tutti i numeri compresi tra **0** e 2^n-1. Complessivamente quindi possiamo specificare gli indirizzi relativi a 2^n celle di memoria distinte; per conoscere la quantita' massima di memoria fisica indirizzabile dalla **CPU** dobbiamo quindi definire l'ampiezza in bit di ogni cella di memoria. Nelle piattaforme hardware basate sulle **CPU 80x86** e' stato deciso di adottare l'ampiezza di **8** bit (**1** byte) per ogni cella di memoria; in definitiva, possiamo dire che con un **Address Bus** a **n** linee, la **CPU** e' in grado di accedere ad un massimo di 2^n byte di memoria fisica.
Le varie linee che compongono l'**Address Bus** vengono indicate per convenzione con i simboli **A0**, **A1**, **A2**, **A3**, etc; il simbolo **A0** indica ovviamente la linea lungo la quale transita il bit meno significativo del numero binario che codifica l'indirizzo a cui vogliamo accedere.
La tabella di Figura 6 illustra una serie di casi che si riferiscono a **CPU** realmente esistenti; il codice **80586** indica in modo generico i vari modelli di **CPU** riconducibili alla classe **Pentium I**:

Figura 6

CPU	Address Bus (bit)	Memoria fisica (byte)
8080	16	65536
8086	20	1048576
80286	24	16777216
80386	32	4294967296
80486	32	4294967296
80586	32	4294967296

Le considerazioni appena esposte si riferiscono all'indirizzamento della memoria centrale del computer; ci si puo' chiedere come avvenga invece l'indirizzamento delle altre periferiche. Come e' stato gia' anticipato, anche i vari dispositivi di **I/O** vengono indirizzati con la stessa tecnica utilizzata per la memoria centrale; ad ogni dispositivo di **I/O** quindi viene assegnato un numero intero che ne rappresenta in modo univoco l'indirizzo. La tastiera del computer ad esempio si trova all'indirizzo esadecimale **60h**; e' compito della **CL** fare in modo che l'indirizzo di una periferica secondaria non venga confuso con l'indirizzo di una cella della memoria centrale. Nel caso in cui la **CPU** debba comunicare con la tastiera, la **CL** abilita questa periferica e disabilita la memoria centrale; in questo modo l'indirizzo **60h** inviato attraverso l'**Address Bus** mette in comunicazione la **CPU** con la tastiera e non con la cella **60h** della memoria centrale.
Ciascuna periferica secondaria e' dotata di una memoria propria che nella gran parte dei casi ammonta a pochi byte; questa piccola area di memoria viene chiamata **porta hardware**. Un normale joystick a due assi e due pulsanti ad esempio utilizza una memoria di appena **1** byte; attraverso questo byte il joystick fornisce alla **CPU** tutte le informazioni relative allo stato (on/off) dei due pulsanti e alla posizione **x**, **y** della leva.
Per indirizzare questa categoria di periferiche secondarie la **CPU** utilizza una parte delle linee dell'**Address Bus**; con le vecchie **CPU** come l'**8086** venivano utilizzate solo le prime **8** linee dell'**Address Bus** (da **A0** a **A7**) con la possibilita' quindi di gestire sino a 2^8**=256** periferiche differenti. A partire dalle **CPU 80386** vengono invece utilizzate le prime **16** linee dell'**Address Bus** (da **A0** a **A15**); in questo caso e' possibile gestire sino a 2^{16}**=65536** periferiche differenti.
Esistono pero' anche periferiche secondarie particolari come la scheda video; le schede video dell'ultima generazione arrivano ad avere una memoria propria (memoria video) che ammonta ormai a qualche centinaio di **Mb**. Questa memoria ovviamente non puo' essere indirizzata con la stessa tecnica delle **porte hardware** utilizzata per un joystick o per un mouse; nel caso della memoria video (e nei casi analoghi) si ricorre allora ad una tecnica che prende il nome di **I/O memory mapped** (input/output mappato nella memoria centrale). Come si intuisce dal nome, questa tecnica consiste nel mappare nella memoria centrale la porzione di memoria della periferica secondaria a cui si vuole accedere; in sostanza, in una precisa area della memoria centrale viene creata una immagine della porzione di memoria della periferica secondaria. Per accedere alla memoria della periferica la **CPU** non deve fare altro che accedere all'immagine presente nella memoria centrale; tutte le operazioni di **I/O** compiute dalla **CPU** su questa immagine si ripercuotono istantaneamente sulla memoria della periferica.

Gli indirizzi di memoria inviati attraverso l'**Address Bus** vengono ricevuti da un apposito circuito decodificatore che provvede a mettere in collegamento la **CPU** con la cella di memoria desiderata; a questo punto puo' iniziare l'interscambio di dati che naturalmente si svolge attraverso il **Data Bus**, evidenziato in Figura 5 con il colore verde.
Le varie linee che compongono il **Data Bus** vengono indicate per convenzione con i simboli **D0**, **D1**, **D2**, **D3**, etc; il simbolo **D0** indica ovviamente la linea lungo la quale passa il bit meno significativo del dato binario che sta transitando sul **Data Bus**.
L'ampiezza del **Data Bus**, e cioe' il numero di linee da cui e' composto, determina l'ampiezza massima in bit che possono avere i dati gestiti via hardware dalla **CPU**; osserviamo infatti che su un **Data Bus** formato ad esempio da **16** linee possono transitare numeri binari formati al massimo da **16** bit.
In base allora a quanto e' stato esposto nei precedenti capitoli, possiamo dire che l'ampiezza del **Data Bus** e' un parametro importantissimo in quanto definisce la cosiddetta **architettura** della **CPU**; una **CPU** con architettura a **n** bit (cioe' con **Data Bus** a **n** linee) e' dotata di **R.C.** in grado di eseguire via hardware operazioni logico aritmetiche su numeri binari a **n** bit.
Nel caso ad esempio della **CPU 80486**, il **Data Bus** a **32** bit permette la gestione via hardware di dati binari formati al massimo da **32** bit; se vogliamo gestire dati binari a **64** bit, dobbiamo procedere via software scrivendo un apposito programma che scompone ogni dato a **64** bit in gruppi

da **32** bit.
Come conseguenza pratica delle cose appena dette, segue anche il fatto che l'ampiezza in bit del **Data Bus** influisce sulle caratteristiche dei vari tipi di dati che possiamo simulare via hardware con la **CPU**; nei precedenti capitoli abbiamo visto infatti che con **n** bit possiamo rappresentare ad esempio tutti i numeri interi senza segno compresi tra:
0 e 2^n-1
e tutti i numeri interi con segno compresi tra:
$-(2^{n-1})$ e $+(2^{n-1}-1)$
La tabella di Figura 7 illustra una serie di casi che si riferiscono a **CPU** realmente esistenti; in questa tabella vengono mostrati i limiti inferiore e superiore dei numeri interi con e senza segno rappresentabili dalla **CPU** in funzione del numero di linee del **Data Bus**. Nel caso della **80586**, il **Data Bus** e' internamente a **64** linee, mentre esternamente alla **CPU** e' formato da **32** linee.

Figura 7

CPU	Data Bus (bit)	Numeri Senza Segno		Numeri Con Segno	
		min.	max.	min.	max.
8080	8	0	255	-128	+127
8086	16	0	65535	-32768	+32767
80286	16	0	65535	-32768	+32767
80386	32	0	4294967295	-2147483648	+2147483647
80486	32	0	4294967295	-2147483648	+2147483647
80586	64	0	18446744073709551615	-9223372036854775808	+9223372036854775807

Come si nota dalla Figura 7, raddoppiando il numero di linee del **Data Bus**, si verifica un enorme aumento dei numeri binari rappresentabili; osservando infatti che con **n** bit possiamo formare 2^n numeri binari diversi e con **2n** bit possiamo formare 2^{2n} numeri binari diversi, possiamo dire che passando da un **Data Bus** a **n** linee ad un **Data Bus** a **2n** linee, i numeri binari rappresentabili crescono di un fattore pari a:
$2^{2n} / 2^n = 2^n$
Naturalmente, tutto cio' comporta un notevole aumento della complessita' circuitale in quanto bisogna ricordare che una **CPU** con **Data Bus** a **n** linee deve essere in grado di manipolare numeri binari a **n** bit e in particolare deve essere in grado di eseguire operazioni logico aritmetiche su numeri binari a **n** bit; aumentare quindi il numero di linee del **Data Bus** significa dover aumentare in modo considerevole le porte logiche delle **R.C.** che eseguono queste operazioni. Il problema fondamentale che i progettisti devono affrontare, consiste nel riuscire ad inserire nello spazio ristretto della **CPU** un numero di componenti elettronici (transistor) che ormai supera abbondantemente i dieci milioni; la **CPU AMD Athlon** ad esempio integra al suo interno circa **22000000** di transistor!

Torniamo ora alla Figura 5 per chiarire il significato dei termini che indicano i vari dispositivi presenti nel computer.

7.4.1 Memoria RAM

Il dispositivo indicato in Figura 5 con la sigla **RAM** rappresenta la memoria centrale del computer; la sigla **RAM** sta per **Random Access Memory** (memoria ad accesso casuale). Il termine casuale si riferisce al fatto che il tempo necessario alla **CPU** per accedere ad una qualsiasi cella di memoria e' indipendente dalla posizione (indirizzo) della cella stessa; la **CPU** quindi per accedere ad una

qualsiasi cella impiega un intervallo di tempo costante. Questa situazione che si presenta nelle memorie ad **accesso casuale** e' opposta al caso delle memorie ad **accesso temporale** (come le memorie a nastro) dove il tempo di accesso dipende dalla posizione in cui si trova il dato; nel caso ad esempio delle memorie a nastro (come le cassette che si utilizzavano con i micro computers **Amiga** e **ZX Spectrum**), per accedere ad un dato registrato nella parte iniziale del nastro e' necessario un tempo brevissimo, mentre per accedere ad un dato registrato nella parte finale del nastro e' necessario un tempo lunghissimo dovuto alla necessita' di far scorrere l'intero nastro.
La memoria **RAM** e' accessibile sia in lettura che in scrittura; come vedremo nel prossimo capitolo, anche le memorie **RAM** vengono realizzate con i transistor. Il contenuto della memoria **RAM** permane finche' il computer rimane acceso; non appena il computer viene spento, viene a mancare l'alimentazione elettrica e il contenuto della **RAM** viene perso.

7.4.2 Memoria ROM

Il dispositivo indicato in Figura 5 con la sigla **ROM** e' una memoria del tutto simile alla **RAM**, e si differenzia per il fatto di essere accessibile solo in lettura; la sigla **ROM** infatti sta per **Read Only Memory** (memoria a sola lettura). Il contenuto della memoria **ROM** viene inserito in modo permanente in un apposito **chip** montato sul computer; questo significa che quando si spegne il computer, la memoria **ROM** conserva le sue informazioni.
All'interno della **ROM** troviamo una serie di programmi scritti in **Assembly** che assumono una importanza vitale per il computer; vediamo infatti quello che succede ogni volta che un **PC** viene acceso. Non appena si accende il computer, una parte della memoria **ROM** contenente svariati programmi viene mappata nella memoria **RAM**; il contatore della **CPU** viene inizializzato con l'indirizzo della **RAM** da cui inizia uno di questi programmi chiamato **POST**. Possiamo dire quindi che all'accensione del computer, la prima istruzione in assoluto eseguita dalla **CPU** e' quella che avvia il programma **POST**; la sigla **POST** sta per **Power On Self Test** (autodiagnosi all'accensione), e indica un programma che esegue una serie di test diagnostici all'accensione del computer, per verificare che tutto l'hardware sia funzionante. Qualsiasi problema incontrato in questa fase, viene segnalato attraverso messaggi sullo schermo o (se il problema riguarda proprio lo schermo) attraverso segnali acustici; se ad esempio si accende il computer con la tastiera staccata (o danneggiata), il **POST** stampa sullo schermo un messaggio del tipo:
```
Keyboard or System Unit Error
```
Se il **POST** si conclude con successo, vengono raccolte una serie di importanti informazioni sull'hardware del computer che vengono poi memorizzate in una apposita area della **RAM** e in una particolare memoria chiamata **CMOS Memory**; tutti questi argomenti vengono trattati in dettaglio nella sezione **Assembly Avanzato**.
Al termine del **POST** vengono inizializzati diversi dispositivi hardware e vengono caricati in **RAM** una serie di "mini programmi" (**procedure**) scritti in **Assembly**, che permettono di accedere a basso livello alle principali risorse hardware del computer; nel loro insieme queste procedure rappresentano il **BIOS** o **Basic Input Output System** (sistema primario per l'input e l'output). Le procedure del **BIOS** consentono ad esempio di formattare hard disk e floppy disk, di inviare dati "grezzi" alla stampante, di accedere alla tastiera, al mouse, etc; tutte queste procedure assumono una importanza fondamentale in quanto vengono utilizzate dai sistemi operativi (**SO**) come il **DOS**, **Windows**, **Linux**, etc, per implementare numerosi servizi che permettono agli utenti di lavorare con il computer.
L'ultimo importantissimo compito svolto in fase di avvio dai programmi contenuti nella **ROM**, consiste nel **bootstrap**, e cioe' nella ricerca di un particolare programma contenuto nel cosiddetto **boot sector** (settore di avvio); il termine **boot sector** indica i primi **512** byte di un supporto di memoria come un floppy disk, un hard disk o un CD-ROM. Nei moderni **PC** il **bootstrap** cerca il **boot sector** scandendo in sequenza il primo lettore floppy disk, il primo lettore CD e infine l'hard disk; se la ricerca fornisce esito negativo, il computer si blocca mostrando un messaggio di errore

che indica l'assenza del **SO**. La ricerca si conclude invece con esito positivo se nei primi **512** byte di uno dei supporti di memoria citati in precedenza viene trovato un programma chiamato **boot loader** (caricatore di avvio); in questo caso il **boot loader** viene caricato in memoria e viene fatto eseguire dalla **CPU**. Il compito fondamentale del **boot loader** consiste nel caricare nella memoria **RAM** il **SO** installato sul computer; a questo punto i programmi di avvio della **ROM** hanno concluso il loro lavoro e il controllo passa al **SO**.

7.4.3 Interfaccia tra la CPU e le periferiche

Analizzando la Figura 5 si nota che mentre ad esempio la **RAM** e' connessa in modo diretto alla **CPU**, altrettanto non accade per molte altre periferiche secondarie; questa situazione e' dovuta al fatto che esistono periferiche di tipo digitale e periferiche di tipo analogico. La **RAM** e' una periferica di tipo digitale in quanto gestisce direttamente segnali logici che rappresentano dati binari; anche la **CPU** e' ovviamente un dispositivo di tipo digitale, e quindi possiamo dire che la **CPU** e la **RAM** parlano lo stesso linguaggio. Tutto cio' rende possibile un interscambio diretto di dati tra **CPU** e **RAM** attraverso il **Data Bus**; questo aspetto e' fondamentale in quanto la **RAM** deve essere in grado di fronteggiare l'elevatissima velocita' operativa della **CPU**.
Esistono invece numerose periferiche di tipo analogico che in quanto tali non possono essere collegate direttamente alla **CPU**; una periferica di tipo analogico parla infatti un linguaggio differente da quello binario parlato dalla **CPU**.

Consideriamo ad esempio un normale joystick dotato di due assi (**x**, **y**) e due pulsanti; lo stato **on/off** di ciascun pulsante viene rappresentato attraverso una tensione elettrica. Possiamo associare il valore di tensione **V0** allo stato **off** e il valore di tensione **V1** allo stato **on**; questi valori analogici di tensione per poter essere letti dalla **CPU** devono essere prima convertiti in livelli logici compatibili con quelli usati dalla **CPU** stessa. Possiamo convertire ad esempio il valore **V0** nel valore di tensione **GND** utilizzato dalla **CPU** per rappresentare il livello logico **0**; analogamente, possiamo convertire il valore **V1** nel valore di tensione **+Vcc** utilizzato dalla **CPU** per rappresentare il livello logico **1**.

Per gli assi del joystick la situazione e' piu' complessa; le coordinate **x**, **y** che esprimono la posizione della leva, sono rappresentate dai valori assunti da due resistenze elettriche variabili (reostati). E' chiaro che la **CPU** non puo' essere in grado di leggere direttamente i valori analogici in **ohm** di queste due resistenze; per poter effettuare questa lettura bisogna prima convertire i due valori di resistenza in due numeri binari.

Un esempio ancora piu' emblematico e' rappresentato dal **modem** che come si sa' viene utilizzato per permettere a due o piu' computers di comunicare tra loro attraverso le linee telefoniche; il problema che si presenta e' dato dal fatto che molte linee telefoniche sono ancora di tipo analogico, e quindi possono accettare solamente segnali elettrici analogici. Per gestire questa situazione viene utilizzato appunto il **modem** che ha il compito di effettuare le necessarie conversioni da digitale ad analogico e viceversa.

In fase di trasmissione il **modem** converte i segnali logici che arrivano dalla **CPU** in segnali analogici da immettere nelle linee telefoniche; in fase di ricezione il **modem** converte i segnali analogici che arrivano dalle linee telefoniche in segnali logici destinati alla **CPU**. La conversione in trasmissione viene chiamata **modulazione**, mentre la conversione in ricezione viene chiamata **demodulazione**; dalla unione e dalla contrazione di questi due termini si ottiene il termine **modem**.

Per effettuare queste conversioni vengono utilizzati diversi metodi; la Figura 8 illustra ad esempio un metodo di conversione che viene chiamato **modulazione di ampiezza**:

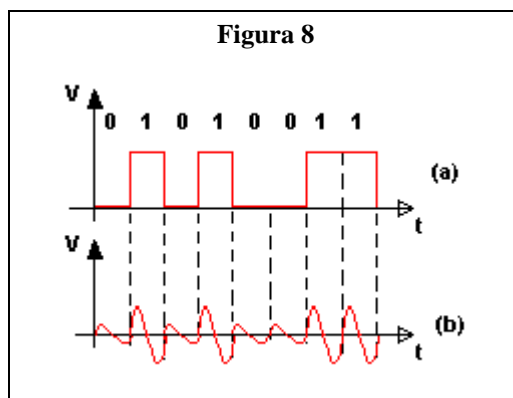



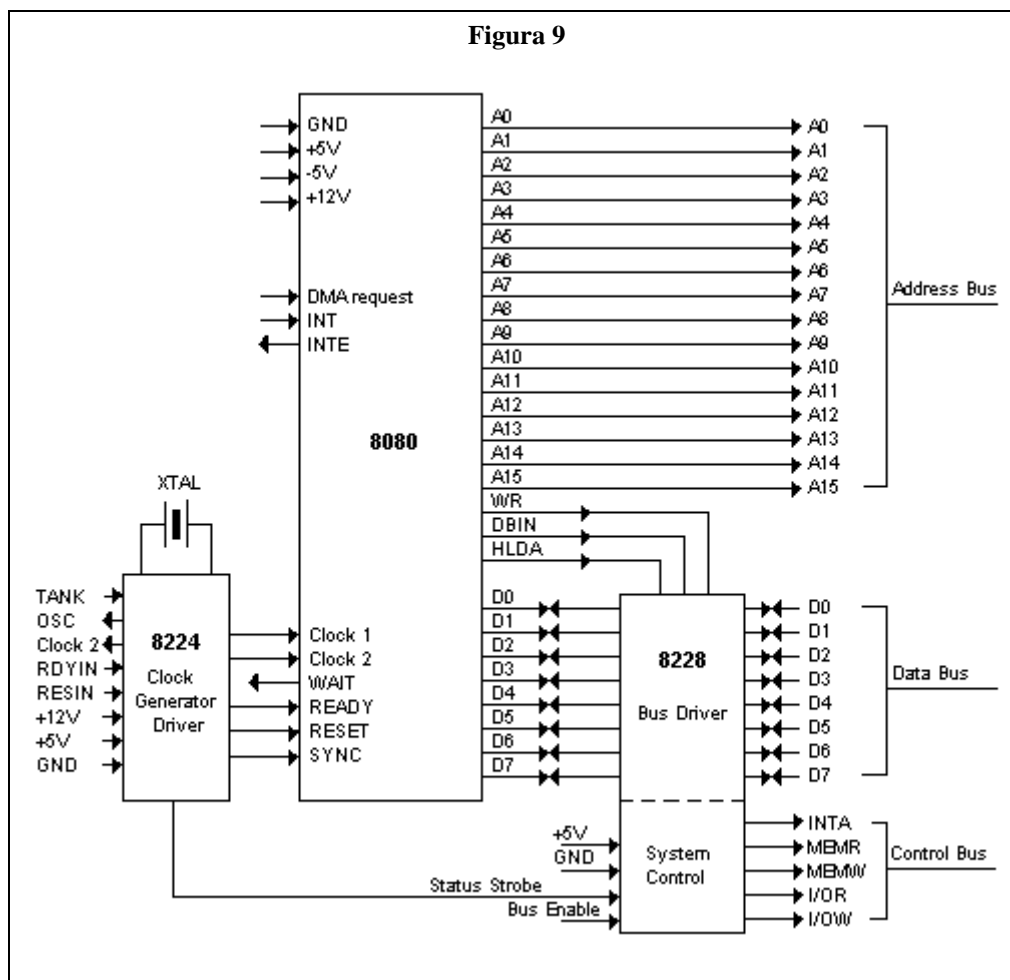
Figura 8

La Figura 2a mostra la rappresentazione logica del numero binario **01010011b**; la Figura 2b mostra invece la rappresentazione analogica dello stesso numero, ottenuta attraverso la modulazione di ampiezza; in pratica, i vari livelli logici che formano il numero binario, vengono convertiti in un segnale elettrico oscillante, con l'ampiezza (altezza) delle varie oscillazioni che e' funzione dei livelli logici stessi. Come si puo' notare, il livello logico **0** viene convertito in una oscillazione avente una determinata ampiezza; il livello logico **1** viene convertito in una oscillazione avente ampiezza maggiore.

Tutte le periferiche di tipo analogico devono essere collegate alla **CPU** attraverso appositi dispositivi che provvedono alle necessarie conversioni da analogico a digitale e viceversa; questi dispositivi vengono genericamente chiamati **interfacce hardware**. L'obiettivo che i produttori di hardware si prefiggono e' quello di realizzare periferiche di tipo digitale che evitando queste conversioni, semplificano enormemente le comunicazioni con la **CPU**; e' chiaro infatti che eliminando la complessa fase di conversione dei segnali, si ottiene un notevole aumento delle prestazioni generali del computer.

7.5 Un esempio reale: la CPU Intel 8080

Per chiarire il significato degli altri dispositivi visibili in Figura 5 e per capire meglio il funzionamento di un computer, analizziamo un esempio reale rappresentato da un computer basato sulla **CPU Intel 8080**; questa **CPU** appartiene ormai alla preistoria, ma come vedremo nei capitoli successivi, il suo "fantasma" aleggia ancora persino sulle **CPU** di classe **Pentium**. La Figura 9 illustra uno schema a blocchi con la cosiddetta **piedinatura** della **CPU 8080**; la piedinatura rappresenta l'insieme dei terminali elettrici che collegano una **CPU** (o un generico **chip**) al mondo esterno:

Figura 9

Osserviamo subito che questa **CPU** e' formata da tre **chip** che rappresentano il generatore di clock **8224**, la **CPU** vera e propria (**8080**) e la logica di controllo **8228**; si nota anche la presenza sui tre **chip** del terminale di massa **GND** e delle tensioni di alimentazione **+5V**, **-5V** e **+12V**.
Come e' stato gia' detto in precedenza, l'**8080** e' una **CPU** con architettura a **8** bit, e dispone di un **Address Bus** a **16** linee con la possibilita' quindi di indirizzare fino a $\mathbf{2^{16}}$**=65536** byte di **RAM** fisica; in Figura 9 si nota infatti in modo evidente la presenza delle **8** linee del **Data Bus** (da **D0** a **D7**), e delle **16** linee dell'**Address Bus** (da **A0** a **A15**).
Nella parte inferiore del lato destro del **chip 8228** (**System Control**) notiamo la presenza del **Control Bus** a **5** linee; su queste **5** linee transitano i segnali di controllo chiamati **INTA**, **MEMR**, **MEMW**, **I/OR** e **IO/W**. La linea **MEMR** abilita una operazione di lettura in **RAM**; la sigla **MEMR** significa infatti **Memory Read**. La linea **MEMW** abilita una operazione di scrittura in **RAM**; la sigla **MEMW** significa infatti **Memory Write**. La linea **I/OR** abilita una operazione di lettura da un dispositivo di **I/O**; la sigla **I/OR** significa infatti **I/O Read**. La linea **I/OW** abilita una operazione di scrittura in un dispositivo di **I/O**; la sigla **I/OW** significa infatti **I/O Write**.

7.5.1 Le interruzioni hardware

Passiamo ora alla linea di controllo **INTA** che svolge un ruolo veramente importante; questa linea opera in combinazione con l'uscita **INTE** e l'ingresso **INT** visibili sul lato sinistro del **chip 8080**. I segnali che transitano su queste linee hanno lo scopo di permettere alle varie periferiche di richiedere la possibilita' di comunicare con la **CPU**; per consentire alle periferiche di poter dialogare con la **CPU**, si possono utilizzare due tecniche principali che vengono chiamate **polling** (sondaggio) e **hardware interrupts** (interruzioni hardware).
Con la tecnica del **polling** la **CPU** interroga continuamente tutte le periferiche per sapere se qualcuna di esse vuole intervenire; l'eventuale richiesta di intervento viene inviata da ogni periferica attraverso appositi segnali logici. Il vantaggio fondamentale della tecnica del **polling** consiste nella sua semplicita' che si traduce quindi in una conseguente semplificazione circuitale; lo svantaggio evidente e' rappresentato invece dal grave rallentamento generale del sistema dovuto al fatto che la **CPU** e' costantemente impegnata nelle operazioni di sondaggio delle varie periferiche. La tecnica del **polling** veniva largamente utilizzata nei vecchi computers; in effetti in passato questa tecnica si rivelava vantaggiosa grazie anche al fatto che le periferiche da collegare ai computers erano molto poche.

Nei moderni computers e in particolare in tutti i **PC** appartenenti alla piattaforma hardware **80x86** viene utilizzata invece la sofisticata tecnica delle **hardware interrupts**; questa tecnica consiste semplicemente nel fatto che se una periferica vuole dialogare con la **CPU**, deve essere essa stessa a segnalare la richiesta.
A tale proposito, su qualsiasi **PC** basato sulle **CPU 80x86** e' presente almeno un **chip** chiamato **8259 PIC**; la sigla **PIC** sta per **Programmable Interrupts Controller** (controllore programmabile delle interruzioni). Il compito fondamentale del **PIC** e' quello di ricevere tutte le richieste di intervento che arrivano dalle varie periferiche; il **PIC** provvede anche a disporre queste richieste in una apposita coda di attesa. Per abilitare la gestione delle **hardware interrupts** la **CPU 8080** si serve dell'apposita linea **INTE** visibile in Figura 9; la sigla **INTE** sta infatti per **interrupts enable** (abilitazione delle interruzioni).
In fase di avvio del computer vengono installate in **RAM** una serie di procedure scritte in **Assembly** che vengono chiamate **ISR**; la sigla **ISR** sta per **Interrupt Service Routine** (procedura per la gestione di una interruzione). Come si deduce dal nome, il compito delle **ISR** e' quello di rispondere alle richieste di intervento che arrivano dalle varie periferiche; gli indirizzi di memoria relativi alle varie **ISR** vengono sistemati in una apposita area della **RAM** e formano i cosiddetti **Interrupt Vectors** (vettori di interruzione). Diverse **ISR** vengono installate da un apposito programma della **ROM**, e rappresentano le procedure del **BIOS** citate in precedenza; numerose altre **ISR** vengono installate dai **SO**.
In base alle considerazioni appena esposte, analizziamo ora quello che succede quando una periferica vuole dialogare con la **CPU**; vediamo in particolare quello che succede quando l'utente preme un tasto sulla tastiera mentre la **CPU** sta eseguendo un programma:

1) Mentre un programma e' in esecuzione, l'utente preme un tasto sulla tastiera.
2) L'hardware della tastiera salva lo Scan Code del tasto in un apposito buffer (area di memoria di una periferica dove si accumulano i dati in transito) e invia un segnale al **PIC**; questo segnale prende il nome di **Interrupt Request** o **IRQ** (richiesta di interruzione).
3) L'**IRQ** inviato dalla tastiera viene ricevuto (insieme ad altre **IRQ** di altre periferiche) dal **PIC**; le varie **IRQ** vengono disposte dal **PIC** nella coda di attesa, e vengono ordinate in base alla loro priorita'. Il **PIC** provvede poi ad inviare alla **CPU** la **IRQ** che si trova in cima alla coda di attesa.
4) La **CPU** riceve la **IRQ** attraverso la linea **INT** visibile sul lato sinistro del **chip 8080** di Figura 9; a questo punto la **CPU** porta a termine l'istruzione che stava elaborando, sospende immediatamente il relativo programma e invia al **PIC** un segnale di via libera; questo segnale viene inviato attraverso

la linea **INTA** (**Interrupt Acknowledgement**) visibile nel **Control Bus** di Figura 9. Il fatto che il programma in esecuzione venga temporaneamente interrotto giustifica la definizione di **interruzione hardware**.
5) Il **PIC** riceve il segnale di via libera e invia alla **CPU** attraverso il **Data Bus** il codice che identifica la periferica che vuole intervenire; questo codice non e' altro che l'indice che individua negli **Interrupt Vectors** l'indirizzo della **ISR** da chiamare.
6) L'**ISR** appena chiamata provvede a soddisfare le richieste della periferica che ha inviato l'**IRQ**; nel caso della pressione di un tasto sulla tastiera, viene chiamata una **ISR** che legge dal buffer della tastiera il codice del tasto premuto e lo mette a disposizione dei programmi. Terminate le varie elaborazioni l'**ISR** segnala alla **CPU** di aver concluso il suo lavoro.
7) La **CPU** provvede a ripristinare il programma precedentemente interrotto; a questo punto la gestione della interruzione e' terminata.

Come si puo' facilmente intuire, il metodo delle **Hardware Interrupts** e' sicuramente piu' complesso della tecnica del **polling**, ma presenta il vantaggio di garantire una gestione enormemente piu' efficiente delle periferiche; e' chiaro infatti che grazie alla tecnica delle interruzioni hardware, la **CPU** dialoga con le periferiche solo quando e' strettamente necessario evitando in questo modo inutili perdite di tempo.

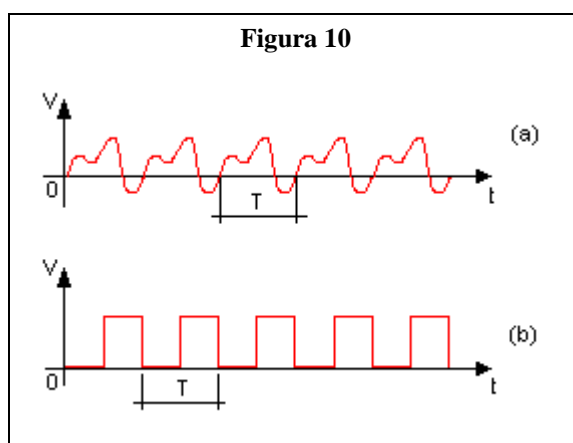
7.5.2 L'accesso diretto alla memoria RAM

Nel caso generale, qualsiasi elaborazione svolta dal computer deve avvenire rigorosamente sotto la supervisione della **CPU**; esiste pero' un caso per il quale l'intervento della **CPU** e' del tutto superfluo e provoca anche un notevole rallentamento delle operazioni. Questo caso si verifica quando dobbiamo semplicemente trasferire un grosso blocco di dati dalla **RAM** ad una periferica di **I/O** o viceversa; in una situazione del genere, non viene eseguita nessuna operazione logico aritmetica, per cui l'intervento della **CPU** e' del tutto inutile.
Queste considerazioni hanno spinto i progettisti dei computers a prevedere l'eventualita' di far comunicare determinate periferiche direttamente con la **RAM**, aggirando in questo modo la **CPU**; l'interscambio diretto di dati tra **RAM** e periferiche rende questa operazione nettamente piu' veloce ed efficiente.
Nelle piattaforme hardware **80x86** l'accesso diretto alla **RAM** da parte delle periferiche viene gestito da un apposito **chip** chiamato **8237 DMAC**; la sigla **DMAC** sta per **Direct Memory Access Controller** (controllore per l'accesso diretto alla memoria). Questo **chip** e' una vera e propria mini **CPU** che e' in grado pero' di eseguire esclusivamente trasferimenti di dati tra **RAM** e periferiche; nei moderni **PC** il **chip 8237** e' stato integrato direttamente nei circuiti del computer.
In Figura 5 vediamo il controllore **DMA** che si trova posizionato nel computer proprio come se fosse una **CPU**; quando una periferica vuole scambiare dati direttamente con la **RAM**, deve comunicarlo al **DMAC**. In altre parole, e' necessario innanzi tutto programmare il **DMAC** in modo da fornirgli le necessarie informazioni che riguardano la periferica che vuole accedere direttamente in **RAM**, quanti byte di dati devono essere trasferiti, etc; il **DMAC** una volta programmato invia la necessaria richiesta alla **CPU** attraverso la linea **DMA Request** visibile in Figura 9 sul lato sinistro del **chip 8080** (questa richiesta viene anche chiamata **HOLD**). Se la **CPU** accetta la richiesta, invia un segnale di via libera attraverso la linea **HLDA** (**HOLD Acknowledgement**) visibile in Figura 9 sul lato destro del **chip 8080**; a questo punto la **CPU** si disattiva e cede il controllo del **Data Bus** al **DMAC**.
Generalmente il **DMAC** suddivide il **Data Bus** in gruppi di **8** linee chiamati **DMA Channels** (canali **DMA**); sulle **CPU** con **Data Bus** a **32** linee ad esempio abbiamo **4** canali **DMA** chiamati: **channel 0** (linee da **D0** a **D7**), **channel 1** (linee da **D8** a **D15**), **channel 2** (linee da **D16** a **D23**) e **channel 3** (linee da **D24** a **D31**).
Solamente alcune periferiche sono predisposte per l'accesso diretto in **RAM**; tra queste periferiche

si possono citare gli hard disk, i lettori di floppy disk, i lettori di CD/DVD e le schede audio. Per avere un'idea della notevole efficienza del **DMA**, si puo' considerare proprio il caso della visione di un film in **DVD** sul computer; in presenza di un lettore **DVD** con supporto per il **DMA** e' possibile avere una riproduzione estremamente fluida del film anche su computers relativamente lenti.
Lavorando sotto **Windows** e' possibile aprire la finestra **Sistema** dalla cartella **Pannello di controllo**, e richiedere la disattivazione del **DMA** sul proprio lettore **DVD**; in questo modo si puo' constatare che la riproduzione dei film in **DVD** subisce un notevole rallentamento con evidenti salti tra un fotogramma e l'altro.

7.5.3 Il segnale di clock

Il funzionamento del computer sarebbe impossibile se non si provvedesse a sincronizzare perfettamente tutti i suoi componenti; la logica di controllo del computer infatti non sarebbe in grado di sapere ad esempio se una determinata operazione aritmetica e' gia' stata eseguita dalla **CPU**, se un dato inviato alla **RAM** e' gia' stato memorizzato, se una periferica ha eseguito un determinato compito, etc.
Per ottenere la sincronizzazione tra tutti i componenti del computer viene utilizzato un apposito segnale al quale viene affidato in pratica il compito di scandire il ritmo di lavoro; i segnali piu' adatti per svolgere questa funzione sono i cosiddetti **segnali periodici**. La Figura 10 illustra due esempi di segnali periodici; in Figura 10a vediamo un segnale periodico di tipo analogico, mentre in Figura 10b vediamo un segnale periodico di tipo logico:

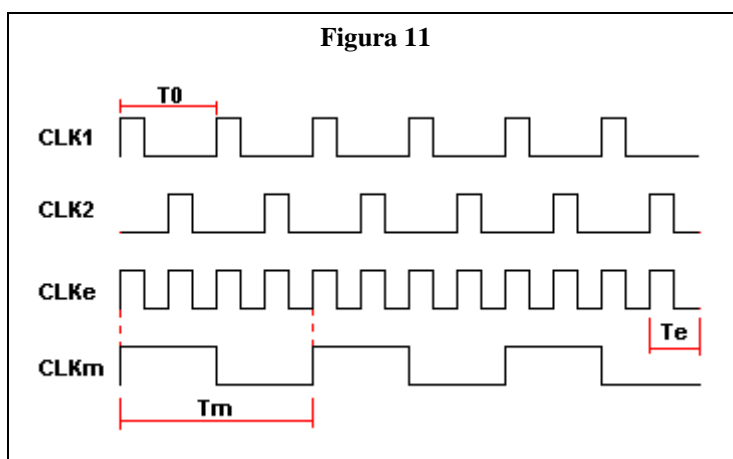
Figura 10

Si definisce **periodico** un segnale i cui valori si ripetono ciclicamente nel tempo; in pratica, un segnale periodico varia assumendo dei valori che si ripetono identicamente ad intervalli regolari di tempo. Ciascuno di questi intervalli regolari di tempo viene chiamato **periodo**; come si vede in Figura 10, il periodo di un segnale periodico viene convenzionalmente indicato con la lettera **T**.
Ciascuna sequenza di valori che si sviluppa in un periodo **T** viene chiamata **ciclo**; il numero di cicli completi che si svolgono nel tempo di **1** secondo rappresentano la **frequenza** del segnale (**f**).
Utilizzando le proporzioni possiamo dire allora che:
`1 ciclo sta a T secondi come f cicli stanno a 1 secondo`
Da questa proporzione si ricava:
`f = 1 / T`
La frequenza quindi e' l'inverso del periodo e di conseguenza la sua unita' di misura e' **sec**$^{-1}$; in onore del fisico tedesco **Hertz** scopritore delle onde radio, l'unita' di misura della frequenza e' stata chiamata **hertz** ed ha per simbolo **Hz**. In elettronica si usano spesso i multipli: **KHz** (migliaia di **Hz**), **MHz** (milioni di **Hz**), **GHz** (miliardi di **Hz**).

La sincronizzazione tra tutti i componenti che formano un computer viene ottenuta attraverso un segnale periodico di tipo logico come quello visibile in Figura 10b; questo segnale viene anche

chiamato **clock** (orologio). Possiamo dire quindi che il segnale di clock di una **CPU** e' un segnale periodico che oscilla continuamente tra il livello logico **0** e il livello logico **1**. La frequenza di oscillazione del segnale di clock di una **CPU** viene definita **frequenza di clock**; come molti sanno, la frequenza di clock e' un parametro fondamentale per una **CPU** in quanto determina la cosiddetta **frequenza di lavoro** della **CPU** stessa. Quando si parla ad esempio di **CPU Athlon** a **1800 MHz**, ci si riferisce al fatto che questa particolare **CPU** riceve un segnale di clock che compie **1800000000** cicli al secondo (**Hz**); se si considera il fatto che tutte le **CPU** a partire dall'**80386** a **33 MHz** sono in grado di compiere una operazione di **I/O** in **RAM** in un solo ciclo di clock, si puo' capire che in linea di principio, maggiore e' la frequenza di clock, maggiore sara' la velocita' operativa del computer.

Il segnale di clock che sincronizza il computer deve avere come caratteristica fondamentale una perfetta stabilita' nel tempo; il modo migliore per ottenere questa caratteristica consiste nello sfruttare una proprieta' naturale del quarzo (e di altri materiali) chiamata **piezoelettricita'**. In pratica, tagliando in modo opportuno un cristallo di quarzo, si ottiene una lamina che sottoposta a pressione lungo una precisa direzione, sviluppa una differenza di potenziale elettrico (tensione); sottoponendo la lamina di quarzo a trazione lungo la stessa direzione, si sviluppa una differenza di potenziale elettrico con polarita' invertite rispetto al caso precedente. In generale, sottoponendo la lamina a continue pressioni e trazioni (vibrazioni), si produce nella lamina stessa una tensione elettrica alternata; l'aspetto importante e' che questo fenomeno e' reversibile, nel senso che sottoponendo la lamina ad una tensione elettrica alternata (la cui polarita' si inverte cioe' ad intervalli di tempo regolari), si produce nella lamina stessa una vibrazione ad altissima frequenza, perfettamente stabile nel tempo. Appositi circuiti elettronici chiamati **oscillatori al quarzo**, convertono queste vibrazioni in un segnale periodico altamente stabile; nel caso del computer, l'oscillatore produrra' un segnale periodico di tipo logico, che oscillera' continuamente tra i valori **0** e **1**, compatibili con i livelli logici della **CPU**. In Figura 5 e in Figura 9 si puo' osservare il simbolo grafico che rappresenta il cristallo di quarzo utilizzato dall'oscillatore; negli schemi elettrici questo cristallo viene sempre indicato con la sigla **XTAL**.

Nel caso piu' generale i computers dispongono di un **clock generator** (generatore di segnale di clock) che produce in uscita un unico segnale di clock da inviare alla **CPU**; in alcuni casi invece il **clock generator** produce due segnali di clock che hanno la stessa frequenza, ma risultano sfasati tra loro. Questo e' proprio quello che accade con la **CPU 8080** visibile in Figura 9; in ogni caso, il segnale di clock singolo o doppio viene utilizzato per ricavare altri segnali periodici destinati a svolgere diversi compiti di sincronizzazione.
Per chiarire meglio questi concetti, facciamo riferimento proprio al doppio segnale di clock prodotto in uscita dal **clock generator 8224** della **CPU 8080** di Figura 9; in questo caso si viene a creare la situazione mostrata in Figura 11:

Figura 11

I due segnali indicati con **CLK1** e **CLK2** vengono generati dal **chip 8224** ed hanno la funzione di segnali di riferimento; come si puo' notare, questi due segnali hanno la stessa frequenza **f0** (e quindi lo stesso periodo **T0**), ma risultano sfasati tra loro di mezzo periodo (**180** gradi). Sovrapponendo questi due segnali (applicandoli ad esempio ad una porta **OR** a due ingressi) la **CPU** ricava un nuovo segnale che in Figura 11 viene indicato con **CLKe**; questo segnale ha quindi frequenza:
`fe = 2 * f0`
e di conseguenza il suo periodo e':
`Te = T0 / 2`
Questo significa che nell'intervallo di tempo impiegato da **CLK1** per compiere un ciclo completo, **CLKe** compie due cicli completi.
CLKe ha un significato molto importante in quanto il suo periodo rappresenta il cosiddetto intervallo di tempo **elementare** della **CPU**; si tratta cioe' dell'intervallo di tempo minimo, necessario alla **CPU** per eseguire le operazioni hardware di base come l'abilitazione della memoria, l'abilitazione di un bus etc. E' chiaro che in teoria sarebbe vantaggioso spingere il piu' in alto possibile il valore della frequenza **fe**; nella pratica pero' bisogna fare i conti con il tempo di risposta delle porte logiche che limita il valore massimo di **fe**.
Sempre dalla Figura 11 si nota che a partire sempre dai due segnali di riferimento, la **CPU** ricava un ulteriore segnale periodico **CLKm**; nel caso di Figura 11 questo segnale ha una frequenza di clock:
`fm = f0 / 2`
e ovviamente, periodo:
`Tm = 2 * T0.`
Anche **CLKm** ha una grande importanza per la **CPU** in quanto ogni suo ciclo rappresenta il cosiddetto **ciclo macchina**, e piu' cicli macchina formano il **ciclo istruzione**, cioe' il numero di cicli macchina necessari alla **CPU** per eseguire una determinata istruzione; ogni istruzione appartenente ad una **CPU** viene eseguita in un preciso numero di cicli macchina, e questo permette alla logica di controllo di sincronizzare tutte le fasi necessarie per l'elaborazione dell'istruzione stessa.

Nei manuali e nella documentazione tecnica relativa ai vari modelli di **CPU**, il numero di cicli del segnale **CLK1** (o **CLK2**) necessari per eseguire una determinata istruzione, viene indicato con la definizione di **clock cycles** (cicli di clock).
Con il passare degli anni il progresso tecnologico sta portando alla realizzazione di dispositivi elettronici sempre piu' veloci che permettono alle nuove **CPU** di lavorare con frequenze di clock sempre piu' elevate; attraverso poi l'individuazione di algoritmi di calcolo sempre piu' efficienti, e' possibile realizzare **R.C.** capaci di eseguire le istruzioni in un numero sempre minore di cicli di clock.
Per fare un esempio pratico, l'**8086** ha una frequenza di clock di circa **5** MHz, ed esegue una moltiplicazione tra numeri interi in oltre **100** cicli di clock pari a:
`100 / 5000000 = 0.00002 secondi = 20000 nanosecondi` (**1** nanosecondo = **1** miliardesimo di secondo).
L'**80486** ha una frequenza di clock di **33** MHz, e per la stessa operazione richiede poco piu' di **10** cicli di clock pari a:
`10 / 33000000 = 0.0000003 secondi = 300 nanosecondi`
In sostanza l'**80486** esegue una moltiplicazione **83** volte piu' velocemente rispetto all'**8086**!

Capitolo 8 - Struttura hardware della memoria

L'evoluzione tecnologica che interessa il mondo del computer, sta assumendo col passare degli anni ritmi sempre piu' vertiginosi; nel corso di questa evoluzione, uno dei componenti che ha acquisito una importanza sempre maggiore e' sicuramente la memoria, al punto che, al giorno d'oggi, non la si considera piu' come una delle tante periferiche, ma come una delle parti fondamentali che insieme alla **CPU** formano il cuore del computer stesso. Lo scopo della memoria e' quello di immagazzinare informazioni in modo temporaneo o permanente; questa distinzione porta a classificare le memorie in due grandi categorie:

Memorie di lavoro
Memorie di massa

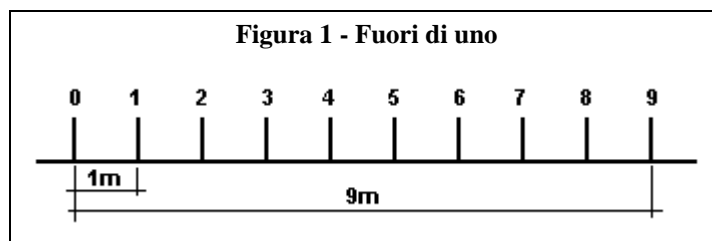
Le memorie di lavoro vengono cosi' chiamate in quanto sono destinate a contenere temporaneamente tutte le informazioni (dati e istruzioni) che devono essere sottoposte ad elaborazione da parte del computer; la fase di elaborazione deve svolgersi nel piu' breve tempo possibile, e quindi e' fondamentale che le memorie di lavoro garantiscano velocita' di accesso elevatissime nelle operazioni di **I/O**. Proprio per questo motivo, queste particolari memorie vengono realizzate mediante dispositivi elettronici a semiconduttore che, come abbiamo visto nel Capitolo 5, presentano tempi di risposta estremamente ridotti; si tenga presente comunque che la velocita' operativa delle memorie di lavoro non puo' certo competere con le prestazioni vertiginose delle attuali **CPU**.

Le memorie di massa hanno lo scopo di immagazzinare in modo permanente grosse quantita' di informazioni che possono essere utilizzate quando se ne ha bisogno; nel momento in cui si ha la necessita' di elaborare queste informazioni, si procede al loro caricamento nella memoria di lavoro. Le memorie di massa vengono realizzate principalmente ricorrendo a supporti ricoperti di materiale magnetico (ossidi di ferro) che attraverso un opportuno circuito elettrico possono essere facilmente magnetizzati (scritti) e smagnetizzati (cancellati). Ad esempio, gli Hard Disk sono costituiti da uno o piu' dischi di alluminio ricoperti di ossido di ferro, mentre i Floppy Disk sono costituiti da un disco di materiale plastico ricoperto sempre da ossido di ferro; uno dei punti di forza dei supporti magnetici e' dato sicuramente dalla enorme capacita' di memorizzazione (decine di Gb), mentre lo svantaggio principale e' dato dalla velocita' di accesso non molto elevata.
Un'altro tipo di memoria di massa che si sta affermando in modo sempre piu' netto e' rappresentato dai **Compact Disk**, scrivibili e riscrivibili attraverso tecnologie laser; attualmente questi tipi di CD sono caratterizzati da discrete velocita' di accesso, mentre la capacita' di immagazzinare informazioni appare ancora insufficiente.

In questo capitolo ci occuperemo delle memorie di lavoro che verranno analizzate dal punto di vista hardware; nel capitolo successivo invece si parlera' della memoria di lavoro dal punto di vista del programmatore.

8.1 Configurazione delle memorie di lavoro

Prima di analizzare la configurazione interna delle memorie di lavoro, e' necessario affrontare un aspetto molto delicato che induce spesso in errore i programmatori meno esperti; questo errore che viene commesso molto frequentemente viene chiamato in gergo: **fuori di uno**. Supponiamo ad esempio di voler recintare un lato di un terreno lungo **9** metri disponendo una serie di paletti ad **1** metro di distanza l'uno dall'altro; la domanda che ci poniamo e': quanti paletti sono necessari? Molte persone rispondono istintivamente **9**, dimenticando cosi' il paletto iniziale che potremmo definire come paletto numero **0**; la Figura 1 chiarisce questa situazione:

Figura 1 - Fuori di uno

Come si puo' notare dalla Figura 1, assegnando l'indice **0** al primo paletto, il secondo paletto avra' indice **1**, il terzo **2**, il quarto **3** e cosi' via, sino all'ultimo paletto (il decimo) che avra' indice **9**; complessivamente abbiamo bisogno di **9** paletti (dal numero **1** al numero **9**) piu' il paletto iniziale (il numero **0**) per un totale di **10** paletti. Questa situazione trae in inganno molte persone che leggendo l'indice **9** sull'ultimo paletto, arrivano alla conclusione che siano presenti solo **9** paletti; questa valutazione errata e' dovuta al fatto che gli indici partono da **0** anziche' da **1**.
E' importantissimo capire questo concetto in quanto nel mondo del computer ci si imbatte spesso in situazioni di questo genere; nei precedenti capitoli abbiamo visto numerosi esempi di sequenze di elementi indicizzati a partire da **0**. Consideriamo ad esempio un computer con **Address Bus** a **16** linee; abbiamo visto che per convenzione la prima linea dell'**Address Bus** viene indicata con **A0** anziche' **A1**, la seconda linea viene indicata con **A1** anziche' **A2** e cosi' via, sino all'ultima linea (la sedicesima) che viene indicata con **A15** anziche' **A16**. Complessivamente abbiamo quindi **15** linee (da **A1** ad **A15**) piu' la linea di indice **0** (**A0**) per un totale di **16** linee.
Con **16** linee possiamo indirizzare 2^{16}**=65536** celle di memoria; se assegnamo alla prima cella l'indirizzo **0**, allora la seconda cella avra' indirizzo **1**, la terza **2**, la quarta **3** e cosi' via, sino all'ultima cella che avra' indirizzo **65535** e non **65536**. Complessivamente possiamo indirizzare **65535** celle (dalla **1** alla **65535**) piu' la cella di indice **0** per un totale di **65536** celle.
Questa convenzione largamente utilizzata nel mondo del computer, permette di ottenere enormi semplificazioni sia software che hardware; osserviamo ad esempio che con il precedente **Address Bus** a **16** linee, la prima cella di memoria si trova all'indirizzo binario **0000000000000000b** (**0d**), mentre l'ultima cella si trova all'indirizzo binario **1111111111111111b** (**65535d**). Se avessimo fatto partire gli indici da **1**, allora la prima cella si sarebbe trovata all'indirizzo binario **0000000000000001b** (**1d**), mentre l'ultima cella si sarebbe trovata all'indirizzo binario **10000000000000000b** (**65536d**); quest'ultimo indirizzo e' formato da **17** bit, e ci costringerebbe ad usare un **Address Bus** a **17** linee!
La situazione appena descritta si presenta spesso anche nella scrittura dei programmi; nel linguaggio **C** ad esempio gli indici dei vettori partono da **0** e non da **1**. Supponiamo allora di avere la seguente definizione:

```
int vett[10]; /* vettore di 10 elementi di tipo int */
```

Il primo elemento di questo vettore e' rappresentato da **vett[0]**, e di conseguenza, l'ultimo elemento (il decimo) e' **vett[9]** e non **vett[10]**; complessivamente abbiamo **9** elementi (da **vett[1]** a **vett[9]**) piu' l'elemento di indice **0** (**vett[0]**) per un totale di **10** elementi.
In definitiva quindi, se abbiamo una generica sequenza di **n** elementi e assegnamo l'indice **0** al primo elemento, allora l'ultimo elemento avra' indice **n-1** e non **n** (l'elemento di indice **n** quindi non esiste); complessivamente avremo **n-1** elementi (dal numero **1** al numero **n-1**) piu' l'elemento di indice **0** per un totale di **n** elementi.

Dopo queste importanti precisazioni, possiamo passare ad analizzare l'organizzazione interna delle memorie di lavoro; tutte le considerazioni che seguono si riferiscono come al solito alle piattaforme hardware basate sulle **CPU** della famiglia **80x86**.
La **CPU** vede la memoria di lavoro come un vettore di celle, cioe' come una sequenza di celle consecutive e contigue; come gia' sappiamo, ciascuna cella ha una ampiezza pari a **8** bit. Nel caso ad esempio di una memoria di lavoro formata da **32** celle, indicando una generica cella con il simbolo C_k (**k** compreso tra **0** e **31**) si verifica la situazione mostrata in Figura 2:

Figura 2 - Vettore di celle di memoria														
Cella	C_0	C_1	C_2	C_3	C_4	C_5	...	C_k	...	C_{26}	C_{27}	C_{28}	C_{29}	C_{30}	C_{31}
Indice	0	1	2	3	4	5	...	k	...	26	27	28	29	30	31

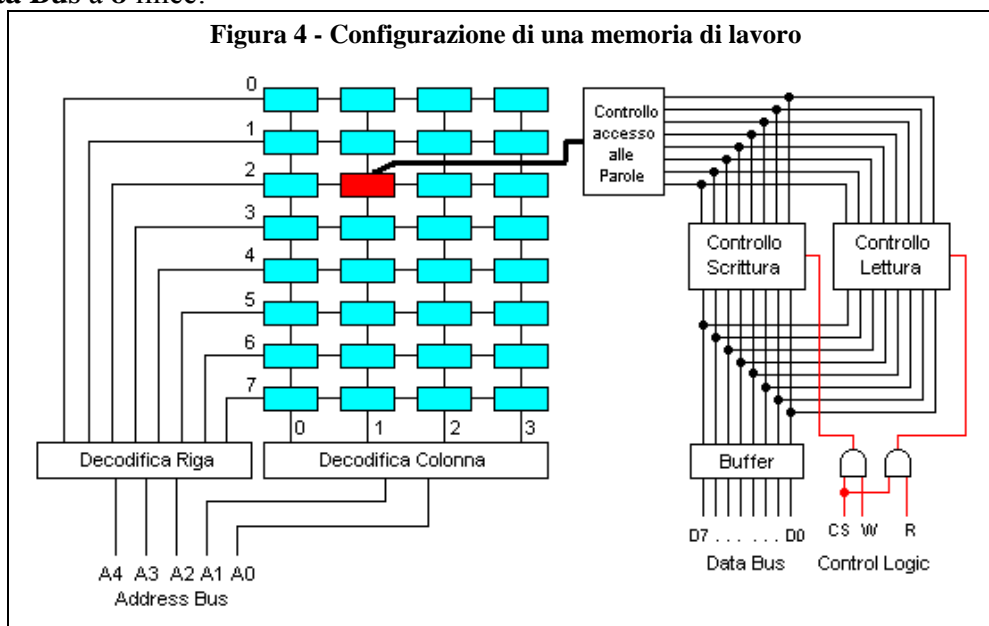
Come si puo' notare, ogni cella viene individuata univocamente dal relativo indice che coincide esattamente con l'indirizzo fisico della cella stessa; se la **CPU** vuole accedere ad una di queste celle, deve specificare il relativo indirizzo che viene poi caricato sull'**Address Bus**. Se vogliamo accedere ad esempio alla trentesima cella, dobbiamo caricare sull'**Address Bus** l'indirizzo **29** (ricordiamoci che gli indici partono da **0**); a questo punto la cella $\mathbf{C_{29}}$ viene messa in collegamento con la **CPU** che puo' cosi' iniziare il trasferimento dati attraverso il **Data Bus**.
Il problema che si presenta e' dato dal fatto che, come e' stato detto nel precedente capitolo, qualunque operazione effettuata dal computer deve svolgesi in un intervallo di tempo ben definito, e cioe' in un numero ben definito di cicli di clock; questo discorso vale naturalmente anche per gli accessi alla memoria di lavoro. In sostanza, l'accesso da parte della **CPU** ad una qualsiasi cella di memoria deve svolgersi in un intervallo di tempo costante, indipendente quindi dalla posizione in memoria (indice) della cella stessa; per soddisfare questa condizione, le memorie di lavoro vengono organizzate sotto forma di matrice di celle. Una matrice non e' altro che una tabella di elementi disposti su **m** righe e **n** colonne; il numero complessivo di elementi e' rappresentato quindi dal prodotto **m*n**. Volendo organizzare ad esempio il vettore di Figura 2 sotto forma di matrice di celle con **8** righe e **4** colonne (**8*4=32**), si ottiene la situazione mostrata in Figura 3:

Figura 3 - Matrice di celle di memoria

Riga	**Colonna**			
	0	**1**	**2**	**3**
0	$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	$C_{0,3}$
1	$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$
2	$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$
3	$C_{3,0}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$
4	$C_{4,0}$	$C_{4,1}$	$C_{4,2}$	$C_{4,3}$
5	$C_{5,0}$	$C_{5,1}$	$C_{5,2}$	$C_{5,3}$
6	$C_{6,0}$	$C_{6,1}$	$C_{6,2}$	$C_{6,3}$
7	$C_{7,0}$	$C_{7,1}$	$C_{7,2}$	$C_{7,3}$

Osserviamo subito che le **8** righe vengono individuate attraverso gli indici da **0** a **7** (indici di riga), mentre le **4** colonne vengono individuate attraverso gli indici da **0** a **3** (indici di colonna); ciascun elemento della matrice viene univocamente individuato attraverso i corrispondenti indice di riga e indice di colonna che costituiscono le coordinate dell'elemento stesso.
Questa organizzazione a matrice permette alla **CPU** di accedere a qualsiasi cella di memoria in un intervallo di tempo costante; la **CPU** non deve fare altro che specificare l'indice di riga e l'indice di colonna della cella desiderata. Questi due indici abilitano la cella che si trova all'incrocio tra la riga e la colonna specificate dalla **CPU**; appare evidente che con questa tecnica, l'accesso ad una cella qualunque richiede un intervallo di tempo costante, e quindi assolutamente indipendente dalla posizione della cella stessa.
In precedenza pero' e' stato detto che la **CPU** vede la memoria sotto forma di vettore di celle; dobbiamo capire quindi come sia possibile ricavare un indice di riga e un indice di colonna da un indirizzo lineare destinato ad un vettore come quello di Figura 2.
A tale proposito osserviamo innanzi tutto che il vettore di Figura 2 puo' essere convertito nella matrice di Figura 3 raggruppando le celle a **4** a **4**; le celle da $\mathbf{C_0}$ a $\mathbf{C_3}$ formano la prima riga della matrice, le celle da $\mathbf{C_4}$ a $\mathbf{C_7}$ formano la seconda riga della matrice e cosi' via. In questo modo e'

semplicissimo passare da una cella della matrice di Figura 3 alla corrispondente cella del vettore di Figura 2; data ad esempio la generica cella $\mathbf{C_{i,j}}$ della matrice di Figura 3, l'indice **k** della corrispondente cella del vettore di Figura 2 si ricava dalla seguente formula:
`k = (i * 4) + j` (**4** rappresenta il numero di colonne della matrice).
Consideriamo ad esempio $\mathbf{C_{2,1}}$ che e' la decima cella della matrice di Figura 3; la corrispondente cella del vettore di Figura 2 e' quella individuata dall'indice:
`(2 * 4) + 1 = 8 + 1 = 9`
Infatti $\mathbf{C_9}$ e' proprio la decima cella del vettore di Figura 2.
Vediamo ora come si procede per la conversione inversa; dato cioe' l'indice **k** di una cella del vettore di Figura 2, dobbiamo ricavare l'indice di riga **i** e l'indice di colonna **j** della corrispondente cella della matrice di Figura 3. Osserviamo subito che per indirizzare **32** celle abbiamo bisogno di un **Address Bus** a $\mathbf{\log_2 32=5}$ linee; con **5** linee infatti possiamo rappresentare $\mathbf{2^5=32}$ indirizzi (da **0** a **31**). Siccome la matrice di Figura 3 e' formata da **8** righe e **4** colonne, suddividiamo le **5** linee dell'**Address Bus** in due gruppi; il primo gruppo e' formato da $\mathbf{\log_2 8=3}$ linee (**A2**, **A3** e **A4**), mentre il secondo gruppo e' formato da $\mathbf{\log_2 4=2}$ linee (**A0** e **A1**). Questi due gruppi di linee ci permettono di individuare l'indice di riga e l'indice di colonna della cella desiderata; per dimostrarlo ripetiamo il precedente esempio. Vogliamo accedere quindi alla cella $\mathbf{C_9}$ del vettore di Figura 2; in binario (a **5** bit) l'indirizzo **9** si scrive **01001b**. Questo indirizzo viene caricato sull'**Address Bus** per cui otteniamo **A0=1**, **A1=0**, **A2=0**, **A3=1**, **A4=0**. Le tre linee da **A2** a **A4** contengono il codice binario **010b** che tradotto in base **10** rappresenta il valore **2**; questo valore rappresenta l'indice di riga. Le due linee da **A0** a **A1** contengono il codice binario **01b** che tradotto in base **10** rappresenta il valore **1**; questo valore rappresenta l'indice di colonna. La cella individuata nella matrice di Figura 3 sara' quindi $\mathbf{C_{2,1}}$; questa cella corrisponde proprio alla cella $\mathbf{C_9}$ del vettore di Figura 2.
Vediamo un ulteriore esempio relativo all'accesso alla cella $\mathbf{C_{31}}$ che e' l'ultima cella del vettore di Figura 2; in binario (a **5** bit) l'indirizzo **31** si scrive **11111b**. Questo indirizzo viene caricato sull'**Address Bus** per cui otteniamo **A0=1**, **A1=1**, **A2=1**, **A3=1**, **A4=1**. Le tre linee da **A2** a **A4** contengono il codice binario **111b** che tradotto in base **10** rappresenta il valore **7**; questo valore rappresenta l'indice di riga. Le due linee da **A0** a **A1** contengono il codice binario **11b** che tradotto in base **10** rappresenta il valore **3**; questo valore rappresenta l'indice di colonna. La cella individuata nella matrice di Figura 3 sara' quindi $\mathbf{C_{7,3}}$; come si puo' notare in Figura 3, questa e' proprio l'ultima cella della matrice.
Traducendo in pratica le considerazioni appena svolte, otteniamo la situazione mostrata in Figura 4; questo circuito si riferisce ad un computer con una memoria di lavoro da **32** byte, **Address Bus** a **5** linee e **Data Bus** a **8** linee:

Figura 4 - Configurazione di una memoria di lavoro

La parte di indirizzo contenuta nelle linee **A2**, **A3** e **A4**, raggiunge il circuito chiamato **Decodifica Riga**; questo circuito provvede a ricavare l'indice di riga della cella da abilitare. La parte di indirizzo contenuta nelle linee **A0** e **A1**, raggiunge il circuito chiamato **Decodifica Colonna**; questo circuito provvede a ricavare l'indice di colonna della cella da abilitare. L'unica cella che viene abilitata e' quella che si trova all'incrocio tra i due indici appena determinati; a questo punto il **Data Bus** viene connesso alla cella abilitata rendendo cosi' possibile il trasferimento dati tra la cella stessa e la **CPU**.
Nella parte destra della Figura 4 si nota la presenza del **Data Bus** a **8** linee e della **logica di controllo** che comprende le tre linee **CS**, **W** e **R**; lungo la linea **CS** transita un segnale chiamato **Chip Select** (selezione **chip** di memoria). Se **CS=1** la memoria di lavoro viene abilitata e la **CPU** puo' comunicare con essa attraverso il **Data Bus**; se **CS=0** la memoria di lavoro e' disabilitata e la **CPU** puo' comunicare con altri dispositivi di **I/O** attraverso il **Data Bus**.
Se viene richiesto un accesso in scrittura (**write**) alla memoria di lavoro, la logica di controllo invia i due segnali **W=1** e **R=0** (con **CS=1**); in questo modo, la porta **AND** di sinistra produce in uscita un livello logico **1** che abilita il **Controllo Scrittura**, mentre la porta **AND** di destra produce in uscita un livello logico **0** che disabilita il **Controllo Lettura**.
Se viene richiesto un accesso in lettura (**read**) alla memoria di lavoro, la logica di controllo invia i due segnali **W=0** e **R=1** (con **CS=1**); in questo modo, la porta **AND** di sinistra produce in uscita un livello logico **0** che disabilita il **Controllo Scrittura**, mentre la porta **AND** di destra produce in uscita un livello logico **1** che abilita il **Controllo Lettura**.

8.1.1 Accesso in memoria con Data Bus a 8 bit

La situazione mostrata dalla Figura 4 e' molto semplice grazie al fatto che l'ampiezza del **Data Bus** (**8** bit) coincide esattamente con l'ampiezza in bit di ogni cella di memoria; in questo caso le **8** linee del **Data Bus** vengono connesse agli **8** bit dell'unica cella di memoria abilitata. Per chiarire meglio questo aspetto osserviamo la Figura 5 che mostra le prime **8** celle del vettore di memoria di Figura 2:

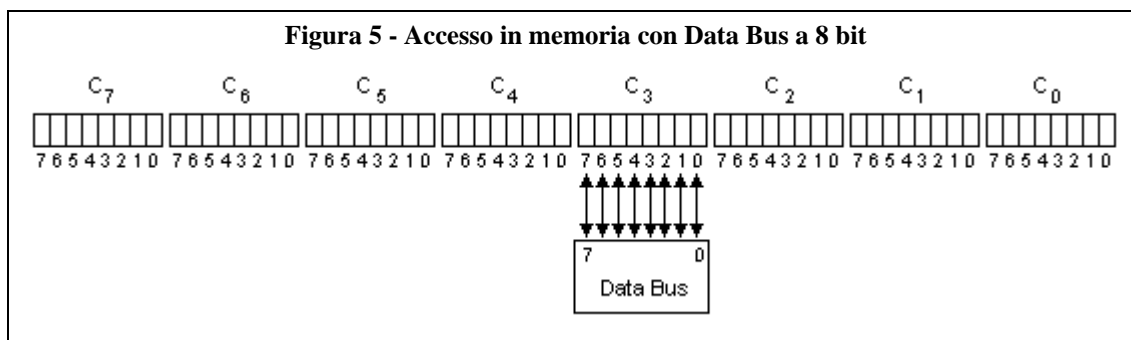
Figura 5 - Accesso in memoria con Data Bus a 8 bit

In questo esempio il **Data Bus** risulta connesso alla cella **C_3** (quarta cella) della memoria di lavoro; com'era prevedibile, la linea **D0** risulta connessa al bit in posizione **0** di **C_3**, la linea **D1** risulta connessa al bit in posizione **1** di **C_3** e cosi' via, sino alla linea **D7** che risulta connessa al bit in posizione **7** di **C_3**. Un **Data Bus** a **8** linee puo' essere posizionato su una qualunque cella secondo lo schema visibile in Figura 5; e' assolutamente impossibile che il **Data Bus** possa posizionarsi a cavallo tra due celle.
Appare evidente il fatto che con un **Data Bus** a **8** linee e' possibile gestire via hardware trasferimenti di dati aventi una ampiezza massima di **8** bit; se vogliamo trasferire un blocco di dati piu' grande di **8** bit, dobbiamo suddividerlo in gruppi di **8** bit. Un dato avente una ampiezza di **8** bit viene chiamato **BYTE** (da non confondere con l'unita' di misura **byte**); il termine **BYTE** definisce quindi un tipo di dato che misura **1** byte, ossia **8** bit.
L'ampiezza in bit del **Data Bus** definisce anche la cosiddetta **parola** del computer; nel caso di

Figura 4 e di Figura 5 abbiamo a che fare quindi con un computer avente **parola** di **8** bit.

8.1.2 Accesso in memoria con Data Bus a 16 bit

Passiamo ora al caso di un computer avente **parola** di **16** bit, cioe' **Data Bus** a **16** linee; il problema che si presenta e' dato dal fatto che la memoria di lavoro e' suddivisa fisicamente in celle da **8** bit, mentre il **Data Bus** ha una ampiezza di **16** bit. Per risolvere questo problema, la memoria di lavoro viene suddivisa logicamente in coppie di celle adiacenti; nel caso ad esempio del vettore di memoria di Figura 2, otteniamo le coppie ($\mathbf{C_0}$, $\mathbf{C_1}$), ($\mathbf{C_2}$, $\mathbf{C_3}$), ($\mathbf{C_4}$, $\mathbf{C_5}$) e cosi' via, sino alla coppia ($\mathbf{C_{30}}$, $\mathbf{C_{31}}$).
Un **Data Bus** a **16** linee puo' essere posizionato su una qualunque di queste coppie; e' assolutamente impossibile quindi che il **Data Bus** possa posizionarsi a cavallo tra due coppie di celle. Per chiarire meglio questo aspetto osserviamo la Figura 6 che mostra le prime **8** celle del vettore di memoria di Figura 2, suddivise in coppie:

Figura 6 - Accesso in memoria con Data Bus a 16 bit

In questo esempio il **Data Bus** risulta connesso alla coppia ($\mathbf{C_2}$, $\mathbf{C_3}$) della memoria di lavoro; le linee da **D0** a **D7** risultano connesse alla cella $\mathbf{C_2}$, mentre le linee da **D8** a **D15** risultano connesse alla cella $\mathbf{C_3}$.
Dalla Figura 6 si deduce che con un **Data Bus** a **16** linee e' possibile gestire via hardware trasferimenti di dati aventi una ampiezza di **8** bit o di **16** bit; se vogliamo trasferire un blocco di dati piu' grande di **16** bit, dobbiamo suddividerlo in gruppi di **8** o **16** bit. In precedenza abbiamo visto che un dato avente una ampiezza di **8** bit viene chiamato **BYTE**; un dato avente ampiezza di **16** bit viene invece chiamato **WORD**. Non bisogna confondere il termine **WORD** con l'unita' di misura **word**; infatti il termine **WORD** indica un tipo di dato che misura **1** word, ossia **2** byte, ossia **16** bit.
Come gia' sappiamo, l'indirizzo di un dato di tipo **BYTE** coincide con l'indirizzo della cella che lo contiene; qual'e' invece l'indirizzo di un dato di tipo **WORD**?
Per rispondere a questa domanda bisogna tenere presente che le **CPU** della famiglia **80x86** nel disporre i dati in memoria seguono una convenzione chiamata **little-endian**; questa convenzione prevede che i dati formati da due o piu' **BYTE** vengano disposti in memoria a con il **BYTE** meno significativo che occupa l'indirizzo piu' basso. Supponiamo ad esempio di avere il dato a **16** bit **0111001100001111b** che si trova in memoria a partire dall'indirizzo **350**; la convenzione **little-endian** prevede che questo dato venga disposto in memoria secondo lo schema mostrato in Figura 7:

Figura 7 - Convenzione "little endian"

Contenuto	0	1	1	1	0	0	1	1		0	0	0	0	1	1	1	1
Posizione bit	7	6	5	4	3	2	1	0		7	6	5	4	3	2	1	0
Indirizzo Cella	351									350							

Come si puo' notare, il **BYTE** meno significativo **00001111b** viene disposto nella cella **350**, mentre il **BYTE** piu' significativo **01110011b** viene disposto nella cella **351**; altre **CPU**, come ad esempio quelle prodotte dalla **Motorola**, utilizzano invece la convenzione inversa (**big-endian**).
Nelle piattaforme hardware **80x86** l'indirizzo di un dato di tipo **WORD** coincide con l'indirizzo della cella che contiene il **BYTE** meno significativo del dato stesso; nel caso di Figura 7 ad esempio, l'indirizzo del dato a **16** bit e' **350**.
La Figura 7 ci permette anche di osservare che se vogliamo tracciare uno schema della memoria su un foglio di carta, ci conviene disporre gli indirizzi in ordine crescente da destra verso sinistra; in questo modo i dati binari (o esadecimali) contenuti nelle varie celle ci appaiono disposti nel verso giusto (cioe' con il peso delle cifre che cresce da destra verso sinistra).

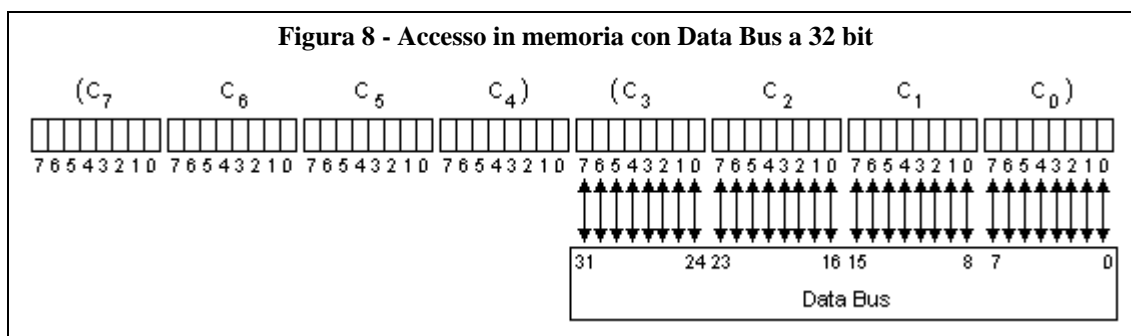
Analizziamo ora come avviene il trasferimento dati di tipo **BYTE** e **WORD** con un **Data Bus** a **16** linee.
In riferimento alla Figura 6, supponiamo di voler leggere il **BYTE** contenuto nella cella C_2; il **Data Bus** viene posizionato sulla coppia (C_2, C_3), e l'accesso alla cella C_2 avviene attraverso le linee da **D0** a **D7**.
Supponiamo di voler leggere il **BYTE** contenuto nella cella C_3; il **Data Bus** viene posizionato sulla coppia(C_2, C_3), e l'accesso alla cella C_3 avviene attraverso le linee da **D8** a **D15**.
Supponiamo di voler leggere il **BYTE** contenuto nella cella C_4; il **Data Bus** viene posizionato sulla coppia (C_4, C_5), e l'accesso alla cella C_4 avviene attraverso le linee da **D0** a **D7**.
Le considerazioni appena svolte ci fanno capire che con un **Data Bus** a **16** linee i dati di tipo **BYTE** non presentano nessun problema di **allineamento** in memoria; questo significa che un dato di tipo **BYTE**, indipendentemente dal suo indirizzo, richiede un solo accesso in memoria per essere letto o scritto.

In riferimento alla Figura 6, supponiamo di voler leggere la **WORD** contenuta nelle celle C_2 e C_3; il **Data Bus** viene posizionato sulla coppia (C_2, C_3), e l'accesso alle celle C_2 e C_3 avviene attraverso le linee da **D0** a **D15**.
Supponiamo di voler leggere la **WORD** contenuta nelle celle C_3 e C_4; il **Data Bus** viene prima posizionato sulla coppia (C_2, C_3), e attraverso le linee da **D8** a **D15** viene letto il **BYTE** che si trova nella cella C_3. Successivamente il **Data Bus** viene posizionato sulla coppia (C_4, C_5), e attraverso le linee da **D0** a **D7** viene letto il **BYTE** che si trova nella cella C_4; complessivamente vengono effettuati due accessi in memoria.
Le considerazioni appena svolte ci fanno capire che con un **Data Bus** a **16** linee i dati di tipo **WORD** non presentano nessun problema di **allineamento** in memoria purche' il loro indirizzo sia un numero pari (**0**, **2**, **4**, **6**, etc); un dato di tipo **WORD** che si trova ad un indirizzo dispari, richiede due accessi in memoria per essere letto o scritto.
Si tenga presente che il disallineamento dei dati in memoria puo' provocare sensibili perdite di tempo anche con le potentissime **CPU** dell'ultima generazione; proprio per evitare questi problemi, l'**Assembly** e molti linguaggi di programmazione di alto livello forniscono ai programmatori tutti gli strumenti necessari per allineare correttamente i dati in memoria.

8.1.3 Accesso in memoria con Data Bus a 32 bit

Anche se il meccanismo dovrebbe essere ormai chiaro, analizziamo un ulteriore caso che si riferisce ad un computer avente **parola** di **32** bit, cioe' **Data Bus** a **32** linee; in questo caso il problema da affrontare riguarda il fatto che la memoria di lavoro e' suddivisa fisicamente in celle da **8** bit, mentre il **Data Bus** ha una ampiezza di **32** bit. Per risolvere questo problema, la memoria di lavoro viene suddivisa logicamente in quaterne di celle adiacenti; nel caso ad esempio del vettore di memoria di Figura 2, otteniamo le quaterne:
(C_0, C_1, C_2, C_3), (C_4, C_5, C_6, C_7) e cosi via, sino alla quaterna (C_{28}, C_{29}, C_{30}, C_{31}).

Un **Data Bus** a **32** linee puo' essere posizionato su una qualunque di queste quaterne; e' assolutamente impossibile quindi che il **Data Bus** possa posizionarsi a cavallo tra due quaterne di celle. Per chiarire meglio questo aspetto osserviamo la Figura 8 che mostra le prime **8** celle del vettore di memoria di Figura 2, suddivise in quaterne:

Figura 8 - Accesso in memoria con Data Bus a 32 bit

In questo esempio il **Data Bus** risulta connesso alla quaterna (**C_0**, **C_1**, **C_2**, **C_3**) della memoria di lavoro; le linee da **D0** a **D7** risultano connesse alla cella **C_0**, le linee da **D8** a **D15** risultano connesse alla cella **C_1**, le linee da **D16** a **D23** risultano connesse alla cella **C_2** e le linee da **D24** a **D31** risultano connesse alla cella **C_3**.
Dalla Figura 8 si deduce che con un **Data Bus** a **32** linee e' possibile gestire via hardware trasferimenti di dati aventi una ampiezza di **8** bit, di **16** bit o di **32** bit; se vogliamo trasferire un blocco di dati piu' grande di **32** bit, dobbiamo suddividerlo in gruppi di **8**, **16** o **32** bit. Un dato avente ampiezza di **32** bit viene chiamato **DWORD** (da non confondere con l'unita' di misura **dword** o **double word**); il termine **DWORD** indica quindi un tipo di dato che misura **1** dword, ossia **2** word, ossia **4** byte, ossia **32** bit.
In base a quanto e' stato detto in precedenza sulla convenzione **little-endian**, l'indirizzo di un dato di tipo **DWORD** coincide con l'indirizzo della cella che contiene il **BYTE** meno significativo del dato stesso; nel caso di Figura 8 ad esempio, un dato di tipo **DWORD** che occupa le celle **C_2**, **C_3**, **C_4** e **C_5** ha indirizzo **2** (cioe' l'indirizzo della cella **C_2**).

Analizziamo ora come avviene il trasferimento dati di tipo **BYTE**, **WORD** e **DWORD** con un **Data Bus** a **32** linee.
In riferimento alla Figura 8, supponiamo di voler leggere il **BYTE** contenuto nella cella **C_0**; il **Data Bus** viene posizionato sulla quaterna (**C_0**, **C_1**, **C_2**, **C_3**), e l'accesso alla cella **C_0** avviene attraverso le linee da **D0** a **D7**.
Supponiamo di voler leggere il **BYTE** contenuto nella cella **C_1**; il **Data Bus** viene posizionato sulla quaterna (**C_0**, **C_1**, **C_2**, **C_3**), e l'accesso alla cella **C_1** avviene attraverso le linee da **D8** a **D15**.
Supponiamo di voler leggere il **BYTE** contenuto nella cella **C_2**; il **Data Bus** viene posizionato sulla quaterna (**C_0**, **C_1**, **C_2**, **C_3**), e l'accesso alla cella **C_2** avviene attraverso le linee da **D16** a **D23**.
Supponiamo di voler leggere il **BYTE** contenuto nella cella **C_3**; il **Data Bus** viene posizionato sulla quaterna (**C_0**, **C_1**, **C_2**, **C_3**), e l'accesso alla cella **C_3** avviene attraverso le linee da **D24** a **D31**.
Supponiamo di voler leggere il **BYTE** contenuto nella cella **C_4**; il **Data Bus** viene posizionato sulla quaterna (**C_4**, **C_5**, **C_6**, **C_7**), e l'accesso alla cella **C_4** avviene attraverso le linee da **D0** a **D7**.
Anche in questo caso si nota subito che con un **Data Bus** a **32** linee i dati di tipo **BYTE** non presentano nessun problema di **allineamento** in memoria; un dato di tipo **BYTE**, indipendentemente dal suo indirizzo, richiede quindi un solo accesso in memoria per essere letto o scritto.

In riferimento alla Figura 8, supponiamo di voler leggere la **WORD** contenuta nelle celle **C_0** e **C_1**; il **Data Bus** viene posizionato sulla quaterna (**C_0**, **C_1**, **C_2**, **C_3**), e l'accesso alle celle **C_0** e **C_1** avviene attraverso le linee da **D0** a **D15**.

Supponiamo di voler leggere la **WORD** contenuta nelle celle $\mathbf{C_1}$ e $\mathbf{C_2}$; il **Data Bus** viene posizionato sulla quaterna ($\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$), e l'accesso alle celle $\mathbf{C_1}$ e $\mathbf{C_2}$ avviene attraverso le linee da **D8** a **D23**.
Supponiamo di voler leggere la **WORD** contenuta nelle celle $\mathbf{C_2}$ e $\mathbf{C_3}$; il **Data Bus** viene posizionato sulla quaterna ($\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$), e l'accesso alle celle $\mathbf{C_2}$ e $\mathbf{C_3}$ avviene attraverso le linee da **D16** a **D31**.
Supponiamo di voler leggere la **WORD** contenuta nelle celle $\mathbf{C_3}$ e $\mathbf{C_4}$; il **Data Bus** viene prima posizionato sulla quaterna ($\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$), e attraverso le linee da **D24** a **D31** viene letto il **BYTE** che si trova nella cella $\mathbf{C_3}$. Successivamente il **Data Bus** viene posizionato sulla quaterna ($\mathbf{C_4}$, $\mathbf{C_5}$, $\mathbf{C_6}$, $\mathbf{C_7}$), e attraverso le linee da **D0** a **D7** viene letto il **BYTE** che si trova nella cella $\mathbf{C_4}$; complessivamente vengono effettuati due accessi in memoria.
Le considerazioni appena svolte ci fanno capire che con un **Data Bus** a **32** linee i dati di tipo **WORD** non presentano nessun problema di **allineamento** in memoria purche' siano interamente contenuti all'interno di una quaterna di celle; se la **WORD** si trova a cavallo tra due quaterne, richiede due accessi in memoria per essere letta o scritta. Un metodo molto semplice per evitare questo problema consiste nel disporre i dati di tipo **WORD** ad indirizzi pari (**0**, **2**, **4**, **6**, etc); in questo modo infatti il dato si viene a trovare o nelle prime due celle o nelle ultime due celle di una quaterna.

In riferimento alla Figura 8, supponiamo di voler leggere la **DWORD** contenuta nelle celle $\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$ e $\mathbf{C_3}$; il **Data Bus** viene posizionato sulla quaterna ($\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$), e l'accesso alle celle $\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$ e $\mathbf{C_3}$ avviene attraverso le linee da **D0** a **D31**.
Supponiamo di voler leggere la **DWORD** contenuta nelle celle $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$ e $\mathbf{C_4}$; il **Data Bus** viene prima posizionato sulla quaterna ($\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$), e attraverso le linee da **D8** a **D31** vengono letti i **3 BYTE** che si trovano nelle celle $\mathbf{C_1}$, $\mathbf{C_2}$ e $\mathbf{C_3}$. Successivamente il **Data Bus** viene posizionato sulla quaterna ($\mathbf{C_4}$, $\mathbf{C_5}$, $\mathbf{C_6}$, $\mathbf{C_7}$), e attraverso le linee da **D0** a **D7** viene letto il **BYTE** che si trova nella cella $\mathbf{C_4}$; complessivamente vengono effettuati due accessi in memoria. La possibilita' di leggere o scrivere un blocco di **3 BYTE** e' una caratteristica delle **CPU Intel** con architettura a **32** bit; in questo modo si riduce a **2** il numero massimo di accessi in memoria per la lettura o la scrittura di un dato di tipo **DWORD**.
Le considerazioni appena svolte ci fanno capire che con un **Data Bus** a **32** linee i dati di tipo **DWORD** non presentano nessun problema di **allineamento** in memoria purche' siano interamente contenuti all'interno di una quaterna di celle; se la **DWORD** si trova a cavallo tra due quaterne, richiede due accessi in memoria per essere letta o scritta. L'unico modo per evitare questo problema consiste nel disporre i dati di tipo **DWORD** ad indirizzi multipli interi di **4** (**0**, **4**, **8**, **12**, etc).

Dopo aver descritto la configurazione interna delle memorie di lavoro, possiamo passare ad analizzare i vari tipi di memoria di lavoro e le tecniche che vengono utilizzate per la fabbricazione dei circuiti di memoria; le memorie di lavoro vengono suddivise in due categorie principali:

Memorie RAM
Memorie ROM

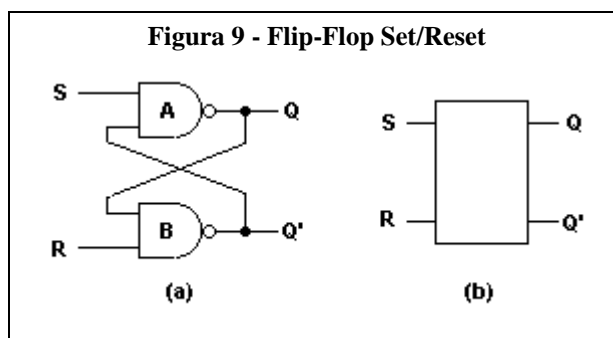
Prima di illustrare la struttura circuitale delle memorie **RAM**, dobbiamo fare la conoscenza con particolari circuiti logici chiamati **reti sequenziali**.

8.2 Le reti sequenziali

Nei precedenti capitoli abbiamo visto che le reti combinatorie sono formate da componenti come le porte logiche che presentano la caratteristica di fornire un segnale in uscita che e' una logica conseguenza del segnale (o dei segnali) in ingresso; se cambiano i livelli logici dei segnali in ingresso, cambia (dopo un piccolo ritardo di propagazione) anche il segnale in uscita, e questo segnale non viene minimamente influenzato dalla precedente configurazione ingresso/uscita assunta dalla porta. In pratica si potrebbe dire che le porte logiche "non ricordano" lo stato che avevano assunto in precedenza; molto spesso pero' si ha la necessita' di realizzare reti combinatorie formate da circuiti logici capaci di ricordare gli stati precedenti, producendo quindi un segnale in uscita che e' una logica conseguenza non solo dei nuovi segnali in ingresso, ma anche degli ingressi precedenti. Le reti combinatorie dotate di queste caratteristiche prendono il nome di **reti sequenziali** e vengono largamente impiegate nella realizzazione delle memorie di lavoro.

8.2.1 Il flip-flop set/reset

Per realizzare un circuito logico capace di ricordare gli stati assunti in precedenza, si sfrutta una tecnica molto usata in campo elettronico, chiamata **retroazione**; questa tecnica consiste nel prelevare il segnale in uscita da un dispositivo, riapplicandolo all'ingresso del dispositivo stesso (o di un'altro dispositivo). Applicando ad esempio la "retroazione incrociata" a due porte **NAND**, si perviene al circuito logico di Figura 9a che rappresenta l'elemento fondamentale delle reti sequenziali e prende il nome di **flip-flop set/reset** (abbreviato in **FF-SR**); la Figura 9b mostra il simbolo del **FF-SR** che si utilizza negli schemi dei circuiti logici.

Figura 9 - Flip-Flop Set/Reset

Per capire il principio di funzionamento del **FF-SR** bisogna ricordare che la porta **NAND** produce in uscita un livello logico **0** solo quando tutti gli ingressi sono a livello logico **1**; in tutti gli altri casi, si otterra' in uscita un livello logico **1**.
Ponendo ora in ingresso **S=1**, **R=1**, si nota che le due uscite **Q** e **Q'** assumono una configurazione del tutto casuale; si puo' constatare pero' che le due uniche possibilita' sono **Q=0**, **Q'=1**, oppure **Q=1**, **Q'=0**. Infatti, se la porta che abbiamo indicato con **A** produce in uscita **1**, attraverso la retroazione questa uscita si porta all'ingresso della porta **B** che avendo entrambi gli ingressi a **1** produrra' in uscita **0**; viceversa, se la porta **A** produce in uscita **0**, questa uscita si porta all'ingresso della porta **B** che avendo in ingresso la configurazione (**0**, **1**) produrra' in uscita **1**.
Partiamo quindi dalla configurazione **R=1**, **S=1**, e tenendo **R=1** portiamo l'ingresso **S** a **0**; osservando la Figura 9a si vede subito che la porta **A**, avendo un ingresso a **0**, produrra' sicuramente in uscita un **1**. La porta **B** trovandosi in ingresso la coppia (**1**, **1**), produrra' in uscita **0**; a questo punto, qualsiasi modifica apportata a **S** (purche' **R** resti a **1**) non modifichera' l'uscita **Q** che continuera' a valere **1**. Abbiamo appurato quindi che partendo dalla configurazione in ingresso (**1**, **1**) e portando in successione **S** prima a **0** e poi a **1**, il **FF-SR** memorizza un bit che vale **1** ed e' disponibile sull'uscita **Q**; questa situazione permane inalterata finche' **R** continua a valere **1**.
Portando ora **R** a **0** e tenendo **S=1**, si vede che la porta **B**, avendo un ingresso a **0** produrra' in uscita

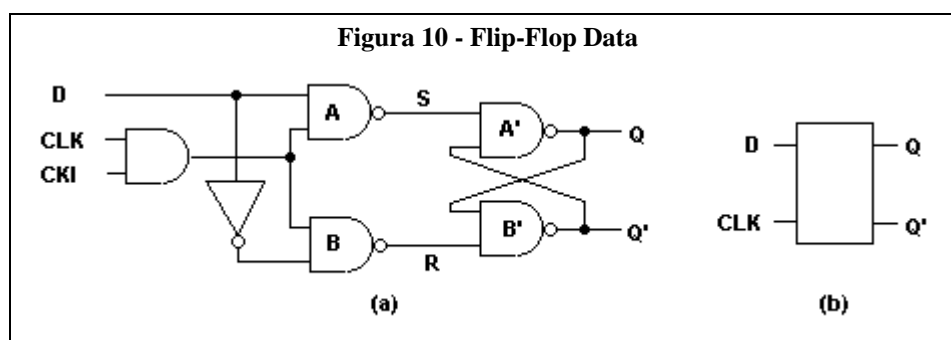
1, mentre la porta **A** ritrovandosi in ingresso la coppia (**1**, **1**) produrra' in uscita **0**; finche' **S** rimane a **1** qualsiasi modifica apportata all'ingresso **R** non altera l'uscita **Q** che continuera' a valere **0**.
Abbiamo appurato quindi che partendo dalla configurazione in ingresso (**1**, **1**) e portando in successione **R** prima a **0** e poi di nuovo a **1**, il **FF-SR** memorizza un bit che vale **0** ed e' disponibile sull'uscita **Q**.

Da tutte le considerazioni appena esposte, risulta evidente che il **FF-SR** rappresenta una unita' elementare di memoria capace di memorizzare un solo bit; infatti:

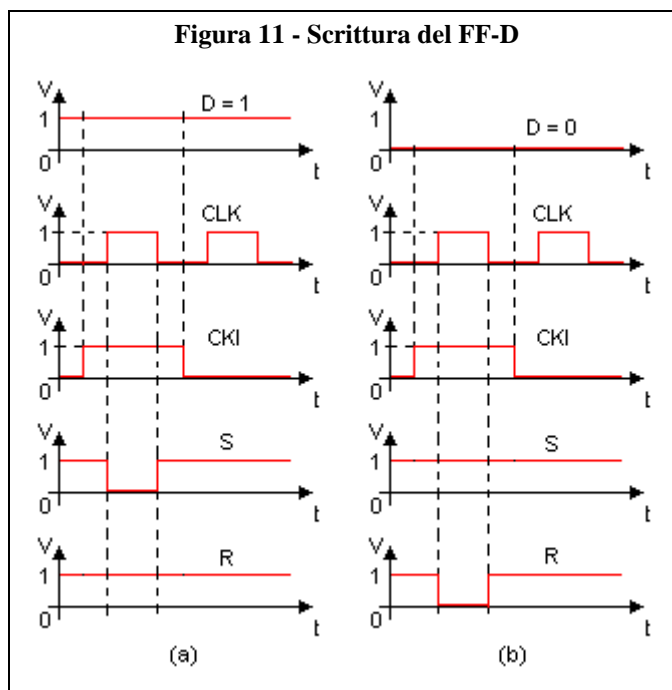
* Attraverso l'ingresso **S** e' possibile scrivere un **1** (**set**) nel **FF-SR**
* Attraverso l'ingresso **R** e' possibile scrivere uno **0** (**reset**) nel **FF-SR**
* Attraverso l'uscita **Q** e' possibile leggere il bit memorizzato nel **FF-SR**.

8.2.2 Il flip-flop data

A partire dal **FF-SR** e' possibile realizzare diversi altri tipi di **flip-flop** che trovano svariate applicazioni nell'elettronica digitale; in particolare, la Figura 10 mostra il **flip-flop** tipo **D** (dove la **D** sta per **Data** o **Delay**), che si presta particolarmente per la realizzazione di memorie di piccole dimensioni. La Figura 10a mostra lo schema circuitale del **flip-flop data** (abbreviato in **FF-D**); la Figura 10b mostra il simbolo logico attraverso il quale si rappresenta il **FF-D**.

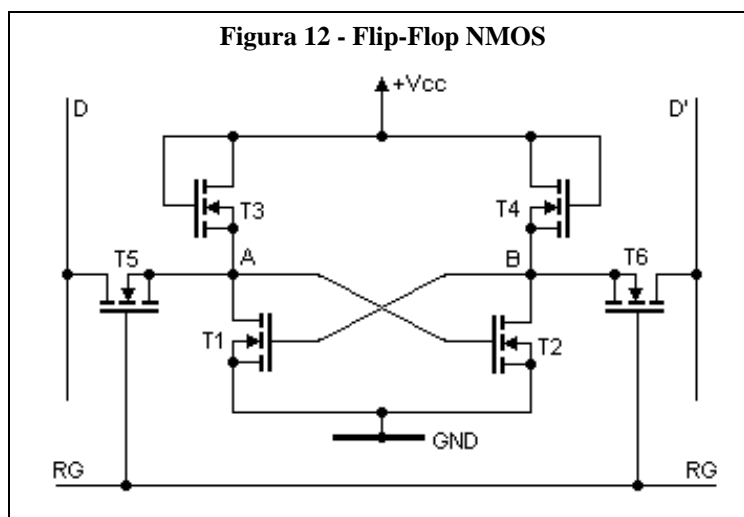
Figura 10 - Flip-Flop Data

L'ingresso **D** viene chiamato **Data**, ed e' proprio da questo ingresso che arriva il bit da memorizzare; l'ingresso **CLK** fornisce al **FF** un segnale di clock (di cui si e' parlato nel precedente capitolo) che consente di sincronizzare le operazioni di **I/O**. In Figura 10a si nota anche un ingresso **CKI** che serve solo per mostrare il principio di funzionamento del **FF-D**, e che ha il compito di impedire (**Clock Inhibit**) o permettere al segnale di clock di arrivare al **FF**; infine le due uscite **Q** e **Q'** sono le stesse del **FF-SR** visto prima.

Per descrivere il principio di funzionamento del **FF-D** notiamo innanzi tutto che, finche' l'ingresso **CKI** e' a **0**, la porta **AND** a sinistra produrra' in uscita un livello logico **0** che andra' a raggiungere entrambe le porte **NAND** indicate con **A** e **B**, che a loro volta produrranno in uscita un livello logico **1**; le due porte **NAND** indicate con **A'** e **B'** costituiscono un **FF-SR** che si trovera' di conseguenza nella configurazione iniziale **S=1**, **R=1**. Partendo da questa configurazione iniziale e abilitando l'ingresso **CKI**, il segnale di clock puo' giungere alle due porte **A** e **B**, e in un ciclo completo (cioe' con il segnale di clock che passa da **0** a **1** e poi di nuovo a **0**) il bit che arriva dall'ingresso **D** verra' copiato sull'uscita **Q**; riportando ora a livello logico **0** l'ingresso **CKI**, il bit appena copiato sull'uscita **Q** restera' "imprigionato" indipendentemente dal valore dell'ingresso **D** (per questo motivo il **FF-D** viene anche chiamato **latch**, che in inglese significa lucchetto).
Per la dimostrazione delle cose appena dette ci possiamo servire della Figura 11: la Figura 11a si riferisce al caso **D=1**, mentre la Figura 11b si riferisce al caso **D=0**.

Figura 11 - Scrittura del FF-D

Cominciamo dal caso in cui dalla linea **D** arrivi un livello logico **1** (Figura 11a); non appena **CKI** passa da **0** a **1**, il segnale di clock viene abilitato e puo' giungere alle due porte **A** e **B**. Analizzando la Figura 10 e la Figura 11a possiamo osservare che quando **CLK** passa da **0** a **1**, le due porte **A** e **B** lasciano **R** a **1** e portano **S** da **1** a **0**; quando **CLK** passa da **1** a **0**, le due porte **A** e **B** lasciano **R** a **1** e portano **S** da **0** a **1**. Applicando allora le cose esposte in precedenza sul **FF-SR** possiamo dire che l'operazione appena effettuata (**set**) ha quindi memorizzato (in un ciclo di clock) l'ingresso **D=1** sul **FF-SR**; il bit memorizzato (**1**) e' disponibile sull'uscita **Q**.
Passiamo ora al caso in cui dalla linea **D** arrivi un livello logico **0** (Figura 11b); non appena **CKI** passa da **0** a **1**, il segnale di clock viene abilitato e puo' giungere alle due porte **A** e **B**. Analizzando la Figura 10 e la Figura 11b possiamo osservare che quando **CLK** passa da **0** a **1**, le due porte **A** e **B** lasciano **S** a **1** e portano **R** da **1** a **0**; quando **CLK** passa da **1** a **0**, le due porte **A** e **B** lasciano **S** a **1** e portano **R** da **0** a **1**. Applicando allora le cose esposte in precedenza sul **FF-SR** possiamo dire che l'operazione appena effettuata (**reset**) ha quindi memorizzato (in un ciclo di clock) l'ingresso **D=0** sul **FF-SR**; il bit memorizzato (**0**) e' disponibile sull'uscita **Q**.

8.2.3 Struttura circuitale di un FF

A titolo di curiosita' analizziamo le caratteristiche costruttive di un **FF** destinato a svolgere il ruolo di unita' elementare di memoria; in Figura 12 vediamo ad esempio un **FF** realizzato in tecnologia **NMOS**.

Figura 12 - Flip-Flop NMOS

Per capire il funzionamento di questo circuito bisogna ricordare che il **MOS** e' un regolatore di corrente pilotato in tensione; regolando da **0** ad un massimo la tensione applicata sul gate, si puo' regolare da **0** ad un massimo la corrente che circola tra il drain e il source. Applicando al gate di un **NMOS** una tensione nulla (o inferiore alla tensione di soglia), il canale tra drain e source e' chiuso e impedisce quindi il passaggio della corrente; in queste condizioni il transistor si comporta come un interruttore aperto (**OFF**). Applicando al gate di un **NMOS** una tensione superiore alla tensione di soglia, il canale tra drain e source e' aperto e favorisce quindi il passaggio della corrente; in queste condizioni il transistor si comporta come un interruttore chiuso (**ON**).

Al centro della Figura 12 si nota il **FF** costituito dai due transistor **T1** e **T2**; i due transistor **T3** e **T4** svolgono semplicemente il ruolo di carichi resistivi per **T1** e **T2**. I due transistor **T5** e **T6** servono per le operazioni di lettura e di scrittura; la linea **RG** infine serve per abilitare la riga lungo la quale si trova la cella (o le celle) a cui vogliamo accedere.
Osserviamo subito che se **T1** e' **ON**, allora **T2** deve essere per forza **OFF**; viceversa se **T1** e' **OFF**, allora **T2** deve essere per forza **ON**. Infatti se **T1** e' **ON**, allora la corrente che arriva dal positivo di alimentazione (**+Vcc**) si riversa a massa proprio attraverso **T1**; questa circolazione di corrente attraverso il carico resistivo **T3** erode il potenziale **+Vcc** portando verso lo **0** il potenziale del punto **A** e quindi anche il potenziale del gate di **T2** che viene spinto cosi' allo stato **OFF**. Analogamente, se **T2** e' **ON**, allora la corrente che arriva dal positivo di alimentazione (**+Vcc**) si riversa a massa proprio attraverso **T2**; questa circolazione di corrente attraverso il carico resistivo **T4** erode il potenziale **+Vcc** portando verso lo **0** il potenziale del punto **B** e quindi anche il potenziale del gate di **T1** che viene spinto cosi' allo stato **OFF**.

Vediamo ora come si svolgono con il circuito di Figura 12 le tre operazioni fondamentali, e cioe': **memorizzazione**, **lettura**, **scrittura**:

Per la fase di memorizzazione (cioe' per tenere il dato in memoria) si pone **RG=0**; in questo modo sia **T5** che **T6** sono **OFF**, per cui il **FF** risulta isolato dall'esterno. Il valore del bit memorizzato nel **FF** e' rappresentato dalla tensione **Vds** che si misura tra il drain e il source di **T1**; se **T1** e' **ON** allora **Vds=0**, mentre se **T1** e' **OFF** allora **Vds=+Vcc**. In base a quanto e' stato detto in precedenza si puo' affermare quindi che la configurazione **T1 ON** e **T2 OFF** rappresenta un bit di valore **0** in memoria; analogamente la configurazione **T1 OFF** e **T2 ON** rappresenta un bit di valore **1** in memoria.

Per la fase di lettura si pone **RG=1**; in questo modo sia **T5** che **T6** sono **ON**. Se il **FF** memorizza un bit di valore **0** allora come gia' sappiamo **T1** e' **ON** e **T2** e' **OFF**; in queste condizioni la corrente che arriva dal positivo di alimentazione transita sia attraverso **T1**, sia attraverso **T5** riversandosi anche lungo la linea **D** che segnala la lettura di un bit di valore **0**. Se il **FF** memorizza un bit di valore **1** allora come gia' sappiamo **T1** e' **OFF** e **T2** e' **ON**; in queste condizioni la corrente che arriva dal positivo di alimentazione transita sia attraverso **T2**, sia attraverso **T6** riversandosi anche lungo la linea **D'** che segnala la lettura di un bit di valore **1**.

Per la fase di scrittura si pone **RG=1**; in questo modo sia **T5** che **T6** sono **ON**. Per scrivere un bit di valore **0** si pone **D=0** e **D'=1**; in questo modo la corrente che arriva dal positivo di alimentazione fluisce verso **D** attraverso **T5**. Il potenziale **+Vcc** viene eroso dal carico resistivo **T3** e quindi il potenziale del punto **A** si porta verso lo zero spingendo **T2** allo stato **OFF** e quindi **T1** allo stato **ON**; come abbiamo visto in precedenza, questa situazione rappresenta un bit di valore **0** in memoria. Per scrivere un bit di valore **1** si pone **D=1** e **D'=0**; in questo modo la corrente che arriva dal positivo di alimentazione fluisce verso **D'** attraverso **T6**. Il potenziale **+Vcc** viene eroso dal carico resistivo **T4** e quindi il potenziale del punto **B** si porta verso lo zero spingendo **T1** allo stato **OFF** e quindi **T2** allo stato **ON**; come abbiamo visto in precedenza, questa situazione rappresenta un bit di valore **1** in memoria.

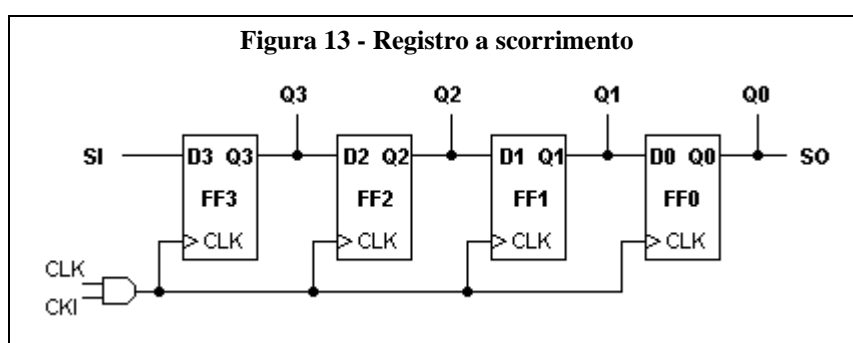
Appare evidente il fatto che il funzionamento dei **FF** e' legato alla presenza della alimentazione elettrica; non appena si toglie l'alimentazione elettrica, il contenuto dei **FF** viene perso. Queste considerazioni si applicano naturalmente a tutte le memorie di lavoro realizzate con i **FF**; possiamo dire quindi che non appena si spegne il computer, tutto il contenuto delle memorie di lavoro di questo tipo viene perso.
Nei paragrafi seguenti vengono utilizzati i **FF** appena descritti per mostrare alcuni esempi relativi ad importanti dispositivi di memoria come i **registri** e le **RAM**.

8.3 Registri di memoria

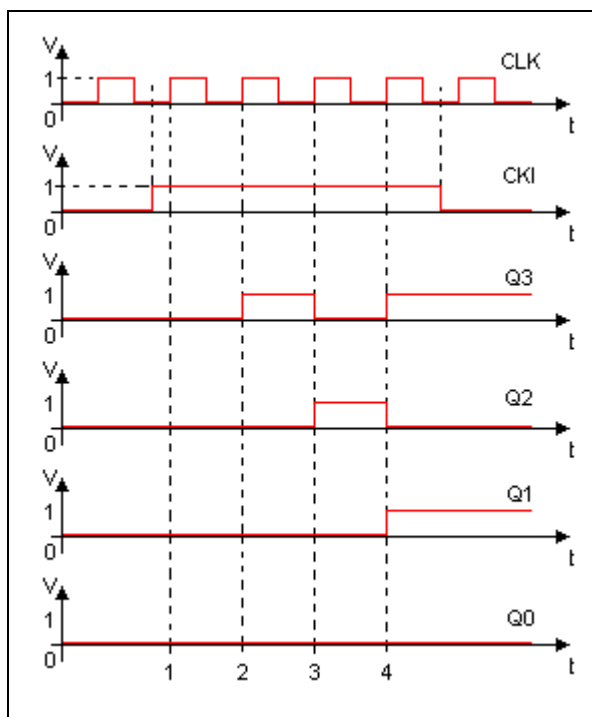
Abbiamo visto che il **FF** rappresenta una unita' elementare di memoria capace di memorizzare un bit di informazione; se si ha la necessita' di memorizzare informazioni piu' complesse costituite da numeri binari a due o piu' bit, bisogna collegare tra loro un numero adeguato di **FF**, ottenendo cosi' particolari circuiti logici chiamati **registri**.

8.3.1 Registri a scorrimento

Collegando piu' **FF** in serie, si ottiene la configurazione mostrata in Figura 13; in particolare, questo esempio si riferisce ad un registro formato da **4 FF-D** che consente di memorizzare numeri a **4** bit (le uscite **Q'** sono state soppresse in quanto non necessarie).

Figura 13 - Registro a scorrimento

Per capire meglio il principio di funzionamento del circuito di Figura 13, bisogna tenere presente che i **FF** vengono realizzati con le porte logiche; come abbiamo visto nei precedenti capitoli, tutte le porte logiche presentano un certo ritardo di propagazione. In sostanza, inviando dei segnali sugli ingressi di una porta logica, il conseguente segnale di uscita viene ottenuto dopo un certo intervallo di tempo; questo ritardo e' dovuto come sappiamo ai tempi di risposta dei transistor utilizzati per realizzare le porte logiche.
Nel circuito di Figura 13 i **4** bit da memorizzare arrivano in serie dall'ingresso **SI** (serial input); in una situazione del genere e' praticamente impossibile quindi memorizzare un nibble in un solo ciclo di clock perche' cio' richiederebbe una risposta istantanea da parte di tutti i **4 FF**. Per risolvere questo problema, la frequenza del segnale di clock viene scelta in modo da minimizzare i tempi di attesa dovuti al ritardo di propagazione; il metodo piu' efficace per ottenere questo risultato consiste nel fare in modo che ad ogni ciclo di clock il registro di Figura 13 sia pronto per ricevere un nuovo bit da memorizzare.

Supponiamo ad esempio di voler memorizzare nel circuito di Figura 13 il numero binario **1010b**; i **4** bit di questo numero arrivano in serie dall'ingresso **SI** a partire dal bit meno significativo. L'ingresso dei vari bit viene sincronizzato attraverso il segnale di clock in modo che ad ogni ciclo di clock arrivi un nuovo bit da memorizzare; analizziamo allora quello che accade attraverso la Figura 14:

Figura 14 - Scrittura del registro

Inizialmente il segnale di clock e' inibito (**CKI=0**) e tutti i **4 FF-SR** contenuti nei **4 FF-D** si troveranno quindi nella situazione iniziale **S=1**, **R=1**; per semplicita' supponiamo che le **4** uscite **Q0**, **Q1**, **Q2** e **Q3** siano tutte a livello logico **0** (anche se cio' non e' necessario).
Come si puo' notare dalla Figura 13, il segnale di clock una volta abilitato (**CKI=1**) giunge contemporaneamente a tutti i **4 FF-D**; a questo punto si verifica la seguente successione di eventi:

* Inizia la fase **1** (Figura 14) con il primo ciclo di clock che raggiunge tutti i **4 FF**.
* Il bit di valore **0** giunge sull'ingresso **D3** di **FF3**.
* Ciascun **FF** copia sulla propria uscita **Q** il segnale che trova sul proprio ingresso **D**.
* Il **FF3** copia il bit **D3=0** sulla sua uscita **Q3**.
* A causa del ritardo di propagazione, **FF2** copia il vecchio **D2=Q3=0** sulla sua uscita **Q2**.
* A causa del ritardo di propagazione, **FF1** copia il vecchio **D1=Q2=0** sulla sua uscita **Q1**.
* A causa del ritardo di propagazione, **FF0** copia il vecchio **D0=Q1=0** sulla sua uscita **Q0**.

* Inizia la fase **2** (Figura 14) con il secondo ciclo di clock che raggiunge tutti i **4 FF**.
* Il bit di valore **1** giunge sull'ingresso **D3** di **FF3**.
* Ciascun **FF** copia sulla propria uscita **Q** il segnale che trova sul proprio ingresso **D**.
* Il **FF3** copia il bit **D3=1** sulla sua uscita **Q3**.
* A causa del ritardo di propagazione, **FF2** copia il vecchio **D2=Q3=0** sulla sua uscita **Q2**.
* A causa del ritardo di propagazione, **FF1** copia il vecchio **D1=Q2=0** sulla sua uscita **Q1**.
* A causa del ritardo di propagazione, **FF0** copia il vecchio **D0=Q1=0** sulla sua uscita **Q0**.

* Inizia la fase **3** (Figura 14) con il terzo ciclo di clock che raggiunge tutti i **4 FF**.
* Il bit di valore **0** giunge sull'ingresso **D3** di **FF3**.
* Ciascun **FF** copia sulla propria uscita **Q** il segnale che trova sul proprio ingresso **D**.
* Il **FF3** copia il bit **D3=0** sulla sua uscita **Q3**.
* A causa del ritardo di propagazione, **FF2** copia il vecchio **D2=Q3=1** sulla sua uscita **Q2**.
* A causa del ritardo di propagazione, **FF1** copia il vecchio **D1=Q2=0** sulla sua uscita **Q1**.
* A causa del ritardo di propagazione, **FF0** copia il vecchio **D0=Q1=0** sulla sua uscita **Q0**.

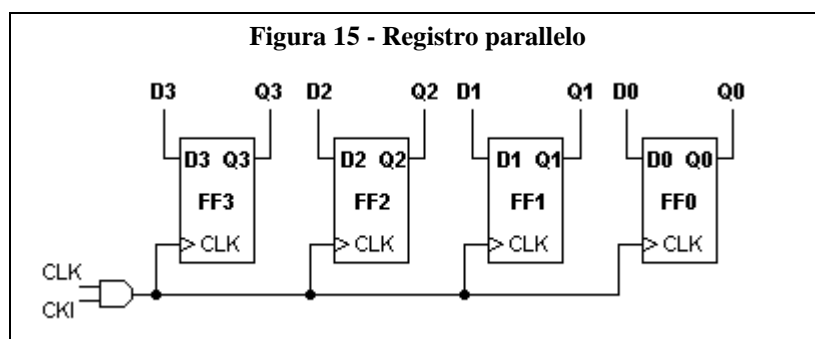
* Inizia la fase **4** (Figura 14) con il quarto ciclo di clock che raggiunge tutti i **4 FF**.

***** Il bit di valore **1** giunge sull'ingresso **D3** di **FF3**.
***** Ciascun **FF** copia sulla propria uscita **Q** il segnale che trova sul proprio ingresso **D**.
***** Il **FF3** copia il bit **D3=1** sulla sua uscita **Q3**.
***** A causa del ritardo di propagazione, **FF2** copia il vecchio **D2=Q3=0** sulla sua uscita **Q2**.
***** A causa del ritardo di propagazione, **FF1** copia il vecchio **D1=Q2=1** sulla sua uscita **Q1**.
***** A causa del ritardo di propagazione, **FF0** copia il vecchio **D0=Q1=0** sulla sua uscita **Q0**.

Come si nota dalla Figura 14, quando **CKI** viene riportato a **0** disabilitando il segnale di clock, si ottiene **Q3=1**, **Q2=0**, **Q1=1**, **Q0=0**; possiamo dire quindi che dopo **4** cicli di clock il circuito di Figura 13 ha memorizzato il nibble **1010b**. Con **CKI=0**, qualunque sequenza di livelli logici in arrivo da **SI** non altera piu' la configurazione assunta dal circuito di Figura 13.
Le considerazioni appena esposte evidenziano il fatto che nel corso dei **4** cicli di clock, i bit da memorizzare scorrono da sinistra verso destra; per questo motivo il circuito di Figura 13 viene anche chiamato **registro a scorrimento**.
Una volta capito il metodo di scrittura nei registri a scorrimento diventa facile capire anche il metodo di lettura; basta infatti inviare altri **4** cicli di clock per far scorrere verso destra i **4** bit precedentemente memorizzati. Questi **4** bit si presentano quindi in sequenza (sempre a partire dal bit meno significativo) sull'uscita **SO** (serial output); dalla Figura 13 si puo' constatare che e' anche possibile leggere i **4** bit direttamente dalle uscite **Q0**, **Q1**, **Q2** e **Q3**.

8.3.2 Registri paralleli

I registri a scorrimento presentano il vantaggio di avere una struttura relativamente semplice in quanto richiedono un numero ridotto di collegamenti; come si nota infatti dalla Figura 13, e' presente un'unica linea per l'input (**SI**) e un'unica linea per l'output (**SO**). Gli svantaggi abbastanza evidenti dei registri a scorrimento sono rappresentati invece dal tempo di accesso in **I/O** e dal metodo di lettura che distrugge il dato in memoria; per quanto riguarda i tempi di accesso in lettura/scrittura si puo' osservare che il numero di cicli di clock necessari e' pari al numero di **FF** che formano il registro. Per quanto riguarda invece l'altro problema, risulta evidente che per leggere ad esempio il contenuto di un registro a **4** bit bisogna inviare **4** cicli di clock; in questo modo escono dalla linea **SO** i **4** bit in memoria, ma dalla linea **SI** entrano altri **4** bit che sovrascrivono il dato appena letto.
Questi problemi vengono facilmente eliminati attraverso il collegamento in parallelo dei **FF**; la Figura 15 mostra appunto un cosiddetto **registro parallelo** costituito anche in questo caso da **4 FF-D**.

Figura 15 - Registro parallelo

Come si puo' notare, ogni **FF** ha l'ingresso **D** e l'uscita **Q** indipendenti da quelli degli altri **FF**; in fase di scrittura, i bit arrivano in parallelo (e tutti nello stesso istante) sugli ingressi **D**, mentre in fase di lettura, i bit vengono letti sempre in parallelo (e tutti nello stesso istante) dalle uscite **Q**.
Ripetendo l'esempio del registro a scorrimento, si vede che in fase di memorizzazione del nibble **1010b**, i **4** bit si presentano contemporaneamente sui **4** ingressi che assumeranno quindi la

configurazione **D3=1**, **D2=0**, **D1=1**, **D0=0**; a questo punto basta inviare un solo ciclo di clock per copiare in un colpo solo i **4** bit sulle rispettive uscite.
Per leggere il dato appena memorizzato, basta leggere direttamente le **4** uscite **Q3**, **Q2**, **Q1**, **Q0**; la lettura avviene senza distruggere il dato stesso.
I registri paralleli risolvono il problema del tempo di accesso e della distruzione dell'informazione in memoria in fase di lettura, ma presentano lo svantaggio di una maggiore complessita' circuitale; un registro parallelo a **n** bit richiedera' infatti solo per l'**I/O** dei dati, **n** linee di ingresso e **n** linee di uscita.

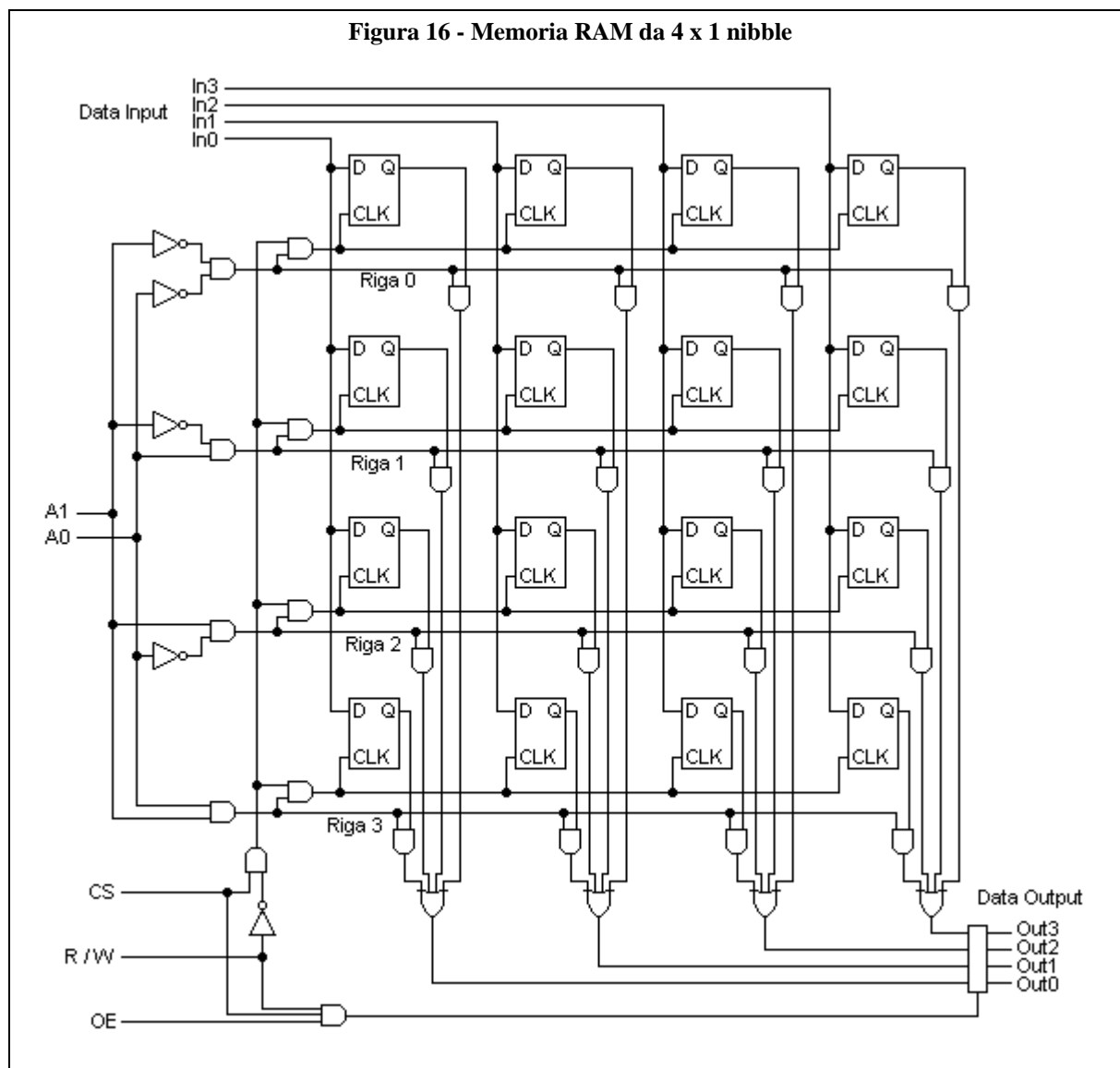
Per le loro caratteristiche i registri vengono largamente utilizzati quando si ha la necessita' di realizzare piccole memorie ad accesso rapidissimo; nel campo dei computers sicuramente l'applicazione piu' importante dei registri e' quella relativa alle memorie interne delle **CPU**. Tutte le **CPU** sono dotate infatti di piccole memorie interne attraverso le quali possono gestire le informazioni (dati e istruzioni) da elaborare; a tale proposito si utilizzano proprio i registri paralleli che garantiscono le massime prestazioni in termini di velocita' di accesso.
Come vedremo nei capitoli successivi, per il programmatore la conseguenza pratica di tutto cio' e' data dal fatto che le istruzioni che coinvolgono dati contenuti nei registri della **CPU** verranno eseguite molto piu' velocemente rispetto al caso delle istruzioni che coinvolgono dati contenuti nella memoria di lavoro; possiamo dire quindi che nei limiti del possibile, il programmatore **Assembly** nello scrivere i propri programmi deve sfruttare al massimo i registri della **CPU**. Questo aspetto e' talmente importante che anche alcuni linguaggi di programmazione di alto livello permettono al programmatore di gestire i registri della **CPU**; il linguaggio **C** ad esempio dispone della parola chiave (qualificatore) **register** attraverso la quale e' possibile richiedere al compilatore l'inserimento di determinati dati nei registri della **CPU**.

8.4 Memorie RAM

L'acronimo **RAM** significa **Random Access Memory** (memoria ad accesso casuale); come gia' sappiamo, il termine "casuale" non si riferisce al fatto che l'accesso a questo tipo di memoria avviene a caso, ma bensi' al fatto che il tempo necessario alla **CPU** per accedere ad una cella qualunque (scelta a caso) e' costante, e quindi assolutamente indipendente dalla posizione in memoria della cella stessa. Al contrario, sulle vecchie unita' di memorizzazione a nastro (cassette), il tempo di accesso dipendeva dalla posizione del dato (memorie ad accesso temporale); piu' il dato si trovava in fondo, piu' aumentava il tempo di accesso in quanto bisognava far scorrere un tratto di nastro sempre piu' lungo.
La **RAM** e' accessibile sia in lettura che in scrittura e il suo ruolo sul computer e' quello di memoria principale (centrale) ad accesso rapido; un programma per poter essere eseguito deve essere prima caricato nella **RAM**, garantendo cosi' alla **CPU** la possibilita' di accedere ai dati e alle istruzioni in tempi ridottissimi. In base a quanto e' stato esposto in precedenza, non appena si spegne il computer tutto il contenuto della **RAM** viene perso; se si vogliono conservare in modo permanente programmi e altre informazioni, bisogna procedere al loro salvataggio sulle memorie di massa.

8.4.1 Memorie RAM statiche (SRAM)

Nella sezione **8.1** di questo capitolo e' stato detto che una memoria di lavoro e' organizzata sotto forma di matrice di celle; se le varie celle vengono realizzate con i **FF**, otteniamo una memoria di lavoro organizzata sotto forma di matrice di registri. Ogni registro deve essere formato naturalmente da un numero fisso di **FF**; nel caso ad esempio delle architetture **80x86**, ogni registro e' formato da **8 FF**.
Analizziamo ora a titolo di curiosita' la struttura circuitale di una **RAM**; la Figura 16 illustra ad esempio una semplicissima **RAM** formata da **4** righe e una sola colonna da **4 FF**.

Figura 16 - Memoria RAM da 4 x 1 nibble

In questo caso essendo presente una sola colonna non abbiamo bisogno del circuito per la **decodifica colonna**; e' presente invece sulla sinistra il circuito per la **decodifica riga**. Per poter selezionare una tra le **4** righe presenti abbiamo bisogno di un **Address Bus** formato da $\mathbf{\log_2 4=2}$ linee (**A0** e **A1**); analizzando il circuito per la **decodifica riga** si puo' constatare facilmente che a seconda dei livelli logici presenti sulle due linee **A0** e **A1** viene selezionata una sola tra le **4** righe presenti.
Supponiamo ad esempio di selezionare la riga **1** (**A0=1**, **A1=0**); in questo caso possiamo osservare che solo sulla linea della riga **1** e' presente un livello logico **1**, mentre sulle altre **3** linee di riga e' presente un livello logico **0**. Se ora vogliamo leggere il nibble memorizzato nei **4 FF** della riga **1**, la logica di controllo pone **CS=1**, **R/W=1**, **OE=1**; sulla linea **CS** transita il segnale **Chip Select** (selezione circuito di memoria), sulla linea **R/W** transita il segnare **Read/Write** (lettura/scrittura), mentre sulla linea **OE** transita il segnale **Output Enable** (abilitazione lettura).
Se proviamo a seguire il percorso dei livelli logici che arrivano da **CS**, **R/W** e **OE** possiamo constatare che agli ingressi **CLK** (clock) di tutti i **FF** arriva un livello logico **0** (clock disabilitato); in questo caso come gia' sappiamo i **FF** non eseguono nessuna operazione di memorizzazione.
Siccome **CS=1**, **R/W=1** e **OE=1**, la porta **AND** piu' in basso nel circuito di Figura 16 produce un livello logico **1** che raggiunge il circuito per l'abilitazione della lettura; in questo modo vengono abilitate le **4** linee del **Data Bus** indicate in Figura 16 con **Data Output**. Dalle **4** uscite **Out0**, **Out1**, **Out2** e **Out3** vengono prelevati i **4** bit memorizzati nei **4 FF** della riga **1**; osserviamo che dalle altre

3 righe vengono letti **3** nibble che valgono tutti **0000b** e vengono poi combinati con il nibble della riga **1** attraverso le porte **OR** a **4** ingressi. Supponiamo che la riga **1** memorizzi il nibble **1101b**; in questo caso sulle **4** linee **Data Output** si ottiene proprio:

```
0000b OR 1101b OR 0000b OR 0000b = 1101b
```

A questo punto la logica di controllo pone **CS=0** disabilitando il chip di memoria; in questo modo si pone fine all'operazione di lettura.

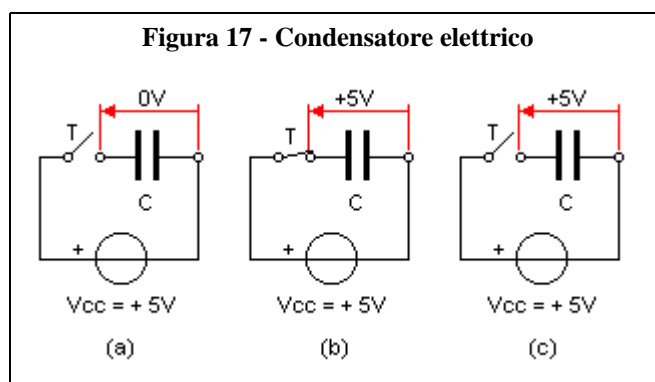
Supponiamo ora di voler scrivere un nibble nei **4 FF** della riga **1**; in questo caso la logica di controllo pone **CS=1**, **R/W=0** e **OE=0** disabilitando cosi' il circuito **Data Output** per la lettura. Se proviamo a seguire il percorso dei livelli logici che arrivano da **CS**, **R/W** e **OE** possiamo constatare che agli ingressi **CLK** (clock) dei **4 FF** della riga **1** arriva un livello logico **1**; sugli ingressi **CLK** dei **FF** di tutte le altre righe arriva invece un livello logico **0** (clock disabilitato). In un caso del genere come gia' sappiamo ciascuno dei **4 FF** della riga **1** copia sulla sua uscita **Q** il segnale presente sul suo ingresso **D** (ricordiamo che con i registri paralleli l'operazione di scrittura richiede un solo ciclo di clock); i **4** segnali che vengono memorizzati sono proprio quelli che arrivano dalle **4** linee del **Data Bus** indicate con **In0**, **In1**, **In2** e **In3** (**Data Input**).
A questo punto la logica di controllo pone **CS=0** disabilitando il chip di memoria; in questo modo si pone fine all'operazione di scrittura.

In base a quanto e' stato appena esposto, possiamo dedurre una serie di considerazioni relative alle memorie **RAM** realizzate con i **FF**; osserviamo innanzi tutto che le informazioni memorizzate in una **RAM** di questo tipo, si conservano inalterate senza la necessita' di ulteriori interventi esterni. Si puo' dire che una **RAM** di questo tipo conservi **staticamente** le informazioni finche' permane l'alimentazione elettrica; proprio per questo motivo, una memoria come quella appena descritta viene definita **Static RAM** (**RAM** statica) o **SRAM**.
Sicuramente il vantaggio fondamentale di una **SRAM** e' rappresentato dai tempi di accesso estremamente ridotti, mediamente dell'ordine di una decina di nanosecondi (**1** nanosecondo = un miliardesimo di secondo = $\mathbf{10^{-9}}$ secondi); questo aspetto e' fondamentale per le prestazioni del computer in quanto buona parte del lavoro svolto dalla **CPU** consiste proprio nell'accedere alla memoria. Se la memoria non e' in grado di rispondere in tempi sufficientemente ridotti, la **CPU** e' costretta a girare a vuoto in attesa che si concludano le operazioni di **I/O**; in un caso del genere le prestazioni generali del computer risulterebbero nettamente inferiori a quelle potenzialmente offerte dal microprocessore.
Purtroppo questo importante vantaggio offerto dalle **SRAM** non riesce da solo a bilanciare i notevoli svantaggi; osservando la struttura della **SRAM** di Figura 16 e la struttura del **FF** di Figura 12, si nota in modo evidente l'enorme complessita' circuitale di questo tipo di memorie. Finche' si ha la necessita' di realizzare **RAM** velocissime e di piccole dimensioni, la complessita' circuitale non rappresenta certo un problema; in questo caso la soluzione migliore consiste sicuramente nello sfruttare le notevoli prestazioni velocistiche dei **FF**. Se pero' si ha la necessita' di realizzare memorie di lavoro di grosse dimensioni (decine o centinaia di Mb), sarebbe teoricamente possibile utilizzare i **FF**, ma si andrebbe incontro a problemi talmente gravi da rendere praticamente irrealizzabile questa idea; si otterrebbero infatti circuiti eccessivamente complessi, con centinaia di milioni di porte logiche, centinaia di milioni di linee di collegamento, elevato ingombro, costo eccessivo, etc.
Proprio per queste ragioni, tutti i moderni **PC** sono dotati di memoria centrale di tipo **dinamico**; come pero' viene spiegato piu' avanti, le **RAM** dinamiche pur essendo nettamente piu' economiche e piu' miniaturizzabili delle **SRAM**, presentano il grave difetto di un tempo di accesso sensibilmente piu' elevato. Con la comparsa sul mercato di **CPU** sempre piu' potenti, questo difetto ha cominciato ad assumere un peso del tutto inaccettabile; per risolvere questo problema, i progettisti dei **PC** hanno deciso allora di ricorrere ad un espediente che permette di ottenere un notevole miglioramento della situazione. A partire dall'**80486** tutte le **CPU** della famiglia **80x86**

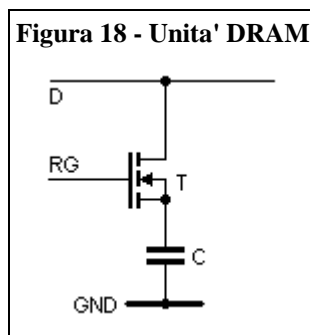
sono state dotate di una piccola **SRAM** interna chiamata **cache memory** (memoria nascondiglio); il trucco che viene sfruttato consiste nel fatto che nell'eseguire un programma presente nella memoria centrale, la **CPU** trasferisce nella **cache memory** le istruzioni che ricorrono piu' frequentemente. Queste istruzioni vengono quindi eseguite in modo estremamente efficiente grazie al fatto che la **cache memory** offre prestazioni nettamente superiori a quelle della memoria centrale.
A causa dello spazio esiguo dovuto alle esigenze di miniaturizzazione, la **cache memory** presente all'interno delle **CPU** e' piuttosto limitata e ammonta mediamente a pochi Kb; questa memoria viene chiamata **L1 cache** o **first level cache** (cache di primo livello). Se si vuole ottenere un ulteriore miglioramento delle prestazioni, e' possibile dotare il computer di una **cache memory** esterna alla **CPU** che viene chiamata **L2 cache** o **second level cache** (cache di secondo livello); tutti i moderni **PC** incorporano al loro interno una **L2 cache** che mediamente ammonta a qualche centinaio di Kb.

8.4.2 Memorie RAM dinamiche (DRAM)

La memoria centrale degli attuali **PC** ammonta ormai a diverse centinaia di Mb; le notevoli dimensioni della memoria centrale devono pero' conciliarsi con il rispetto di una serie di requisiti che riguardano in particolare la velocita' di accesso, il costo di produzione e il livello di miniaturizzazione. Come abbiamo appena visto, le **SRAM** offrono prestazioni particolarmente elevate in termini di velocita' di accesso, ma non possono soddisfare gli altri requisiti; proprio per questo motivo, gli attuali **PC** vengono equipaggiati con un tipo di memoria centrale chiamato **Dynamic RAM** o **DRAM** (**RAM** dinamica).
In una **DRAM** vengono sfruttate le caratteristiche dei condensatori elettrici; si definisce condensatore elettrico un sistema formato da due conduttori elettrici tra i quali si trova interposto un materiale isolante. Ciascuno dei due conduttori elettrici viene chiamato **armatura**; in Figura 17 vediamo ad esempio un condensatore (**C**) formato da due armature piane tra le quali si trova interposta l'aria.

Figura 17 - Condensatore elettrico

In Figura 17a notiamo che se l'interruttore **T** e' aperto, non puo' circolare la corrente elettrica, e quindi il generatore non puo' caricare il condensatore; in queste condizioni tra le due armature del condensatore si misura una tensione elettrica nulla (**0V**).
In Figura 17b notiamo che se l'interruttore **T** viene chiuso, la corrente elettrica puo' circolare e quindi l'energia elettrica erogata dal generatore comincia ad accumularsi nel condensatore; alla fine del processo di carica, tra le due armature del condensatore si misura una tensione elettrica pari a quella fornita dal generatore (**+5V**).
In Figura 17c notiamo che se l'interruttore **T** viene riaperto, il condensatore rimane isolato dall'esterno e quindi teoricamente dovrebbe conservare il suo stato di carica; anche in questo caso tra le due armature del condensatore si misura una tensione elettrica di **+5V**. Se vogliamo scaricare il condensatore, dobbiamo collegarlo ad esempio a una resistenza elettrica; in questo modo la resistenza assorbe l'energia elettrica dal condensatore e la converte in energia termica.

Dalle considerazioni appena esposte si deduce immediatamente che anche il condensatore rappresenta una semplicissima unita' elementare di memoria capace di gestire un singolo bit; possiamo associare infatti il livello logico **0** al condensatore scarico (**0V**), e il livello logico **1** al condensatore carico (**+Vcc**). Mettendo in pratica questi concetti si perviene al circuito di Figura 18 che illustra in linea di principio la struttura di una unita' elementare di memoria di tipo **DRAM**.

Figura 18 - Unita' DRAM

Come si puo' notare, l'unita' elementare di memoria e' composta da un transistor **T** di tipo **NMOS** e da un condensatore **C**; il confronto tra la Figura 18 e la Figura 12 evidenzia la notevole semplicita' di una unita' **DRAM** rispetto a una unita' **SRAM**. La linea indicata con **D** rappresenta la **linea dati** utilizzata per le operazioni di **I/O**, mentre la linea **RG** serve per abilitare o disabilitare la riga su cui si trova la cella desiderata; come al solito si assume che il potenziale di massa (**GND**) sia pari a **0V**.
Se la linea **RG** si trova a livello logico **0**, si vede chiaramente in Figura 18 che il transistor **T** si porta allo stato **OFF** (interdizione); in queste condizioni il condensatore **C** risulta isolato dall'esterno per cui teoricamente dovrebbe conservare indefinitamente il bit che sta memorizzando. Come e' stato detto in precedenza, se il condensatore e' scarico sta memorizzando un bit di valore **0**; se invece il condensatore e' carico sta memorizzando un bit di valore **1**.
Per effettuare una operazione di lettura o di scrittura bisogna portare innanzi tutto la linea **RG** a livello logico **1** in modo che il transistor **T** venga portato allo stato **ON** (saturazione); a questo punto il condensatore puo' comunicare direttamente con la linea **D** per le operazioni di **I/O**.
L'operazione di scrittura consiste semplicemente nell'inviare un livello logico **0** o **1** sulla linea **D**; ovviamente questo livello logico rappresenta il valore del bit che vogliamo memorizzare. Ponendo **D=0**, il condensatore si ritrova con entrambe le armature a potenziale **0V**; in queste condizioni il condensatore si scarica (se non era gia' scarico) e memorizza quindi un bit di valore **0**. Ponendo invece **D=1**, il condensatore si ritrova con una armatura a potenziale **+Vcc** e l'altra a potenziale di massa **0V**; in queste condizioni il condensatore si carica (se non era gia' carico) e memorizza quindi un bit di valore **1**.
L'operazione di lettura con l'unita' **DRAM** di Fugura 18 consiste semplicemente nel misurare il potenziale della linea **D**; se il potenziale e' **0V** (condensatore scarico) si ottiene la lettura **D=0**, mentre se il potenziale e' **+Vcc** (condensatore carico) si ottiene la lettura **D=1**.
Come e' stato gia' evidenziato, l'unita' elementare **DRAM** risulta notevolmente piu' semplice dell'unita' elementare **SRAM** rendendo quindi possibile la realizzazione di memorie di lavoro di notevoli dimensioni, caratterizzate da una elevatissima densita' di integrazione e da un costo di produzione estremamente ridotto; per quanto riguarda l'organizzazione interna, le **DRAM** vengono strutturate sotto forma di matrice rettangolare di celle esattamente come avviene per le **SRAM**.
Dalle considerazioni appena esposte sembrerebbe che le **DRAM** siano in grado di risolvere tutti i problemi presentati dalle **SRAM**, ma purtroppo bisogna fare i conti con un aspetto piuttosto grave che riguarda i condensatori; in precedenza e' stato detto che un condensatore carico isolato dal mondo esterno conserva indefinitamente il suo stato di carica. Un condensatore di questo genere viene chiamato **ideale** ed ha un significato puramente teorico; in realta' accade che a causa degli inevitabili fenomeni di dispersione elettrica, un condensatore come quello di Figura 18 si scarica nel giro di poche decine di millisecondi. Per evitare questo problema tutte le memorie **DRAM**

necessitano di un apposito dispositivo esterno che provvede ad effettuare periodicamente una operazione di rigenerazione (**refreshing**) delle celle; nelle moderne **DRAM** questo dispositivo viene integrato direttamente nel chip di memoria.
Al problema del refreshing bisogna aggiungere il fatto che le operazioni di carica e scarica dei condensatori richiedono tempi relativamente elevati; la conseguenza negativa di tutto cio' e' data dal fatto che le **DRAM** presentano mediamente tempi di accesso pari a **60**, **70** nanosecondi. Una **CPU** come l'**80486 DX** a **33** MHz e' in grado di effettuare una operazione di **I/O** in memoria in appena **1** ciclo di clock, pari a:

```
1 / 33000000 = 0.00000003 secondi = 30 nanosecondi
```

Questo intervallo di tempo e' nettamente inferiore a quello richiesto da una **DRAM** da **60** nanosecondi; per risolvere questo problema, i progettisti dei **PC** hanno escogitato i cosiddetti **wait states** (stati di attesa). In pratica, osservando che **60** nanosecondi sono il doppio di **30** nanosecondi, la **CPU** predispone la fase di accesso in memoria in un ciclo di clock, dopo di che gira a vuoto per un altro ciclo di clock dando il tempo alla memoria di rispondere; questa soluzione appare piuttosto discutibile visto che, considerando il fatto che la **CPU** dedica buona parte del suo tempo agli accessi in memoria, ci si ritrova con un computer che lavora prevalentemente a velocita' dimezzata (nel caso dell'**80486** si ha infatti: **33/2=16.5** MHz).
Come e' stato detto in precedenza, questo ulteriore problema viene parzialmente risolto attraverso l'uso della **cache memory** di tipo **SRAM**; l'idea di fondo consiste nel creare nella **cache** una copia delle informazioni che si trovano nella **DRAM**, e che vengono accedute piu' frequentemente dalla **CPU**. La **cache** ha la precedenza sulla **RAM** per cui, ogni volta che la **CPU** deve accedere ad una informazione, se quell'informazione e' presente nella cache l'accesso avviene in un solo ciclo di clock; in caso contrario, l'informazione viene letta dalla **DRAM** con conseguente ricorso agli stati di attesa.

8.5 Memorie ROM

L'acronimo **ROM** significa **Read Only Memory** (memoria a sola lettura); come dice il nome, queste memorie sono accessibili solo in lettura in quanto il loro scopo e' quello di conservare dati e istruzioni la cui modifica potrebbe compromettere il funzionamento del computer.
Il contenuto di queste memorie viene spesso impostato in fase di fabbricazione, e permane anche in assenza di alimentazione elettrica; tutto cio' potrebbe sembrare impossibile visto che anche le **ROM** sono memorie elettroniche che per funzionare hanno bisogno di essere alimentate elettricamente. La conservazione delle informazioni all'interno di una **ROM** viene ottenuta attraverso svariati metodi; uno di questi metodi viene illustrato dalla Figura 19 che mostra una semplicissima **ROM** formata da **4** transistor di tipo **BJT**:

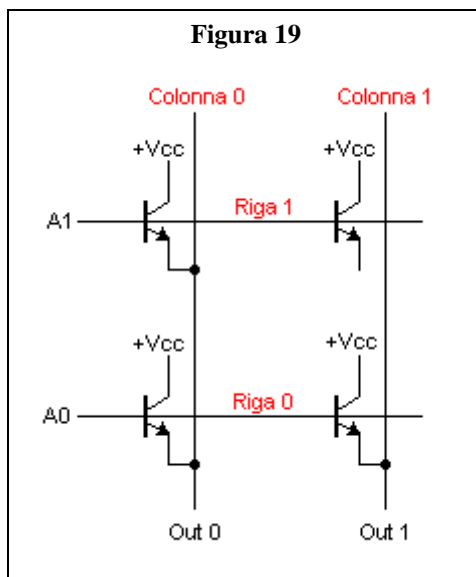



Figura 19

La **ROM** di Figura 19 e' formata da due linee di riga e due linee di colonna; le due righe sono collegate alle linee **A0** e **A1** dell'**Address Bus**, mentre le due colonne forniscono in uscita i livelli logici **Out0** e **Out1** che, come vedremo in seguito, sono una logica conseguenza dei valori assunti da **A0** e **A1**.
Ricordando quanto e' stato detto nel Capitolo 5 sul **BJT**, se sulla base arriva un livello logico **0**, il transistor si porta in interdizione (**OFF**); il potenziale **+Vcc** (livello logico **1**) presente sul collettore non puo' trasferirsi sull'emettitore, e quindi sull'uscita di emettitore avremo un livello logico **0**. Se invece sulla base arriva un livello logico **1**, il transistor si porta in saturazione (**ON**); il potenziale **+Vcc** (livello logico **1**) presente sul collettore si trasferisce sull'emettitore, e quindi sull'uscita di emettitore avremo un livello logico **1**.
In Figura 19 si nota anche la presenza di un transistor con l'emettitore non collegato alla linea di colonna; questi transistor forniscono alla corrispondente colonna sempre un livello logico **0**, e quindi memorizzano di fatto un bit di valore **0**.
A questo punto possiamo capire il principio di funzionamento della memoria **ROM** di Figura 19; i livelli logici assunti dai due ingressi **A0** e **A1** determinano in modo univoco i livelli logici ottenibili sulle uscite **Out0** e **Out1**. Inviando ad esempio sull'**Address Bus** l'indirizzo **10b** (cioe' **A0=0** e **A1=1**), si ottiene in uscita **Out0=1** e **Out1=0**; questi due livelli logici che si ottengono in uscita sono legati univocamente all'indirizzo in ingresso **10b**.
Possiamo dire quindi che in realta' la memoria **ROM** mostrata in Figura 19 non contiene nessuna informazione al suo interno; queste informazioni si vengono a creare automaticamente nel momento in cui la **ROM** viene alimentata e indirizzata.
La memoria appena descritta rappresenta una **ROM** propriamente detta; la struttura interna delle **ROM** propriamente dette viene impostata una volta per tutte in fase di fabbricazione, e non puo' piu' essere modificata. Come abbiamo visto nel precedente capitolo, sui **PC** le **ROM** vengono utilizzate per memorizzare dati e programmi di vitale importanza per il corretto funzionamento del computer; in particolare si possono citare le procedure del **BIOS**, il programma per il **POST** e il programma per il **bootstrap**.

Oltre alle **ROM** propriamente dette, esistono anche diverse categorie di **ROM** che possono essere programmate; analizziamo brevemente i principali tipi di **ROM** programmabili:

PROM o **Programmable ROM** (**ROM** programmabili). Le **PROM** sono molto simili alle **ROM** propriamente dette, ma si differenziano per il fatto di contenere al loro interno dei microfusibili (uno per ogni transistor) posti tra l'uscita di emettitore e la corrispondente colonna; attraverso una apposita apparecchiatura e' possibile far bruciare alcuni di questi microfusibili in modo da impostare la struttura interna della **PROM** a seconda delle proprie esigenze. I microfusibili vengono bruciati (e quindi interrotti) attraverso una corrente pulsante di circa **100 mA**; da queste considerazioni risulta chiaramente il fatto che le **PROM** possono essere programmate una sola volta.

EPROM o **Erasable Programmable ROM** (**ROM** programmabili e cancellabili). Nelle **EPROM** i **BJT** vengono sostituiti da transistor **MOS** che contengono al loro interno anche un secondo gate chiamato **gate fluttuante**; questo secondo gate viene immerso in uno strato di biossido di silicio (**SiO2**), e quindi si trova ad essere elettricamente isolato. Attraverso appositi impulsi di tensione applicati tra drain e source, e' possibile determinare per via elettrostatica un accumulo di carica elettrica nel gate fluttuante; la carica cosi' accumulata e' isolata dall'esterno, e quindi puo' conservarsi per decine di anni. I gate fluttuanti elettricamente carichi rappresentano un bit di valore **1**, mentre quelli scarichi rappresentano un bit di valore **0**.
Il contenuto delle **EPROM** puo' essere cancellato attraverso esposizione ai raggi ultravioletti; per questo motivo, i contenitori che racchiudono le **EPROM** sono dotati di apposite finestre trasparenti che favoriscono il passaggio degli ultravioletti. Una volta che la **EPROM** e' stata cancellata puo'

essere riprogrammata con la tecnica descritta in precedenza; generalmente sono possibili una cinquantina di riprogrammazioni.

EAROM o **Electrically Alterable ROM** (**ROM** alterabili elettricamente). L'inconveniente principale delle **EPROM** descritte in precedenza e' dato dal fatto che queste memorie per poter essere cancellate devono essere rimosse dal circuito in cui si trovano inserite; per ovviare a questo inconveniente sono state create appunto le **EAROM**. Le **EAROM** sono del tutto simili alle **EPROM**, ma si differenziano per la presenza di appositi circuiti che attraverso l'utilizzo di impulsi elettrici, permettono sia la programmazione che la cancellazione della memoria; possiamo dire quindi che le **EAROM** possono essere considerate delle vere e proprie **RAM** non volatili, cioe' delle **RAM** che conservano il loro contenuto anche in assenza di alimentazione elettrica.
L'evoluzione delle **EAROM** ha portato alla nascita delle **EEPROM** chiamate anche **E^2PROM** (**Electrically Erasable PROM**); nelle **EEPROM** lo strato isolante attorno al gate fluttuante viene ridotto a pochi centesimi di micron facilitando in questo modo le operazioni di scrittura e di cancellazione.

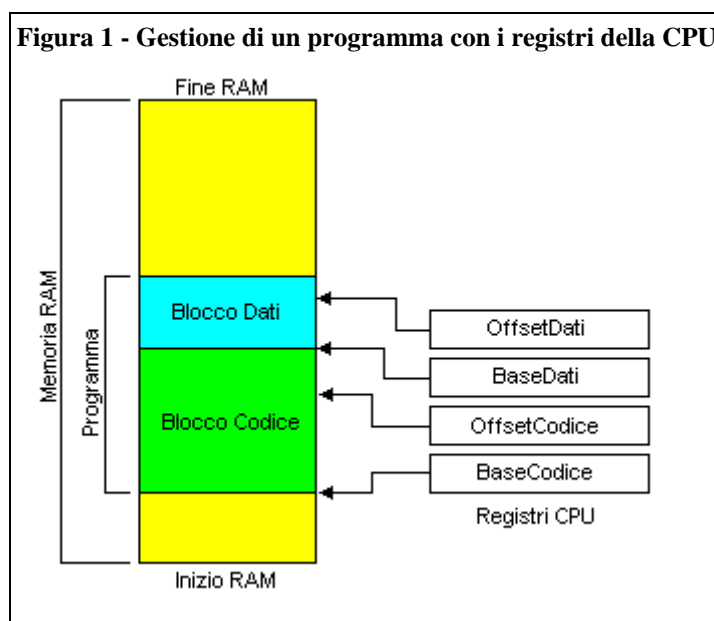
NOVRAM o **Non Volatile RAM** (**RAM** non volatili). Le **NOVRAM** possono essere definite come le **ROM** dell'ultima generazione; queste memorie vengono ottenute associando una **RAM** con una **EEPROM** aventi la stessa capacita' di memorizzazione in byte. In fase operativa, tutte le operazioni di **I/O** vengono effettuate ad alta velocita' sulla **RAM**; nel momento in cui si vuole spegnere l'apparecchiatura su cui e' installata la **NOVRAM**, tutto il contenuto della **RAM** viene copiato in modo permanente nella **EEPROM**.

Capitolo 9 - La memoria dal punto di vista del programmatore

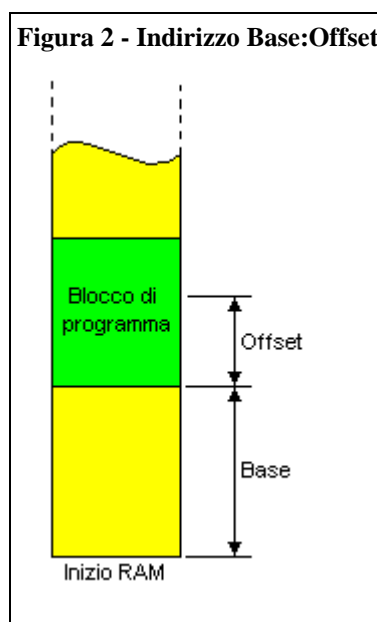
In un precedente capitolo e' stato detto che un insieme di dati e istruzioni, nel loro insieme formano un programma; nel caso piu' semplice e intuitivo quindi, un programma da eseguire con il computer viene suddiviso in due blocchi fondamentali chiamati:

Blocco Dati
Blocco Codice

Il blocco dati e' un'area riservata, ai dati in input del programma, ai dati temporanei e ai dati che scaturiscono dalle varie elaborazioni (dati in output); il blocco codice e' invece un'area riservata alle istruzioni che devono elaborare i dati stessi.
Un programma, per poter essere eseguito, deve essere prima caricato nella memoria **RAM** del computer; in questo modo, la **CPU** puo' accedere rapidamente al programma stesso, e puo' quindi gestire al meglio la fase di elaborazione. Per poter svolgere il proprio lavoro, la **CPU** ha bisogno di conoscere una serie di informazioni sul programma da eseguire; in particolare, la **CPU** ha bisogno di sapere, da quale indirizzo parte il blocco codice, da quale indirizzo parte il blocco dati, qual'e' l'indirizzo della prossima istruzione da eseguire e qual'e' l'indirizzo degli eventuali dati a cui l'istruzione da eseguire fa' riferimento. Tutte queste informazioni vengono gestite dalla **CPU** attraverso i propri registri interni; come e' stato detto nel precedente capitolo, per poter ottenere le massime prestazioni possibili, la **CPU** si serve di registri di tipo parallelo.
Grazie ai propri registri interni, la **CPU** ha, istante per istante, il controllo totale sul programma in esecuzione; la Figura 1 illustra lo schema tipico attraverso il quale una generica **CPU** gestisce l'elaborazione di un programma in memoria.

Figura 1 - Gestione di un programma con i registri della CPU

In questa figura, il blocco colorato in giallo rappresenta la **RAM** del computer; come si puo' notare, la memoria e' disposta dal basso (inizio **RAM**) verso l'alto (fine **RAM**). All'interno della **RAM** e' presente il programma da eseguire; la parte colorata in verde rappresenta il blocco codice, mentre la parte colorata in celeste rappresenta il blocco dati.
Per poter avere il pieno controllo su questo programma, la **CPU** memorizza tutte le necessarie informazioni nei suoi registri interni; la Figura 1 illustra, in particolare, la gestione delle informazioni fondamentali riguardanti, il blocco codice e il blocco dati. Il metodo piu' semplice ed efficiente che permette alla **CPU** di gestire questa situazione, consiste nel rappresentare un qualsiasi

indirizzo, sotto forma di una coppia **Base**, **Offset**, che viene indicata convenzionalmente con il simbolo **Base:Offset**; questo concetto viene illustrato dalla Figura 2, e assume una importanza capitale nel campo della programmazione dei computers.

Figura 2 - Indirizzo Base:Offset

Come si puo' notare, il termine **Base** indica l'indirizzo della cella di memoria da cui inizia un determinato blocco di programma (codice o dati); si tratta chiaramente di un indirizzo **assoluto**, calcolato cioe' rispetto all'inizio della **RAM**. Il termine **Offset**, indica invece l'indirizzo di una cella di memoria interna allo stesso blocco di programma; questo indirizzo viene chiaramente calcolato rispetto a **Base**, ed e' quindi un indirizzo **relativo**, che in inglese viene appunto chiamato **offset** (spiazzamento). La componente **Offset** ci permette di muoverci agevolmente all'interno del blocco di programma che vogliamo indirizzare; possiamo paragonare quindi **Offset** all'indirizzo di una sorta di cursore che si sposta all'interno di un blocco di programma.
Dalle considerazioni appena esposte, si deduce facilmente che una coppia **Base:Offset**, rappresenta univocamente la cella di memoria che si trova all'indirizzo fisico:

```
Base + Offset
```

Ogni volta che il programmatore specifica nei propri programmi un indirizzo **Base:Offset**, la **CPU** calcola la somma tra **Base** e **Offset**, ottenendo cosi' un indirizzo fisico da caricare sull'**Address Bus**; come e' stato detto nel precedente capitolo, questo indirizzo non e' altro che l'indice di una cella appartenente al vettore che forma la **RAM**.

Sulla base di queste considerazioni, osserviamo in Figura 1 che in uno dei registri della **CPU**, viene memorizzato l'indirizzo della **RAM** da cui inizia il blocco codice; questo indirizzo e' stato chiamato simbolicamente **BaseCodice** (indirizzo di partenza del blocco codice).
In un altro registro della **CPU**, viene memorizzato l'indirizzo della **RAM** in cui si trova la prossima istruzione da eseguire; questo indirizzo e' stato chiamato simbolicamente **OffsetCodice** (spiazzamento all'interno del blocco codice).
Mentre **BaseCodice** e' un indirizzo fisso, **OffsetCodice** viene continuamente aggiornato dalla logica di controllo della **CPU**; infatti, nel momento in cui la **CPU** sta eseguendo una istruzione, **OffsetCodice** viene posizionato sulla prossima istruzione da eseguire.
Supponiamo ad esempio di avere una **CPU** con **Address Bus** a **16** bit, e supponiamo inoltre che il **BaseCodice** del programma in esecuzione, sia espresso dal valore esadecimale **0AB2h**; tutto cio' significa che il blocco codice del programma, inizia dalla cella n. **0AB2h** della **RAM**. Se l'**OffsetCodice** della prossima istruzione da eseguire e' **00F4h**, allora l'indirizzo effettivo di questa istruzione e' dato da:

`0AB2h + 00F4h = 0BA6h`

Possiamo dire quindi che il codice macchina della prossima istruzione da eseguire, inizia dalla cella n. **0BA6h** della **RAM**; per poter accedere a questa cella, la **CPU** deve caricare l'indirizzo fisico **0BA6h** sull'**Address Bus**.
In definitiva, la somma:

`BaseCodice + OffsetCodice`

fornisce alla **CPU** l'indirizzo effettivo della cella della **RAM** da cui inizia il codice macchina della prossima istruzione da eseguire; possiamo paragonare **OffsetCodice** all'indirizzo di una sorta di cursore che si muove all'interno del blocco codice del programma in esecuzione.

Tornando alla Figura 1, possiamo notare che in un altro registro della **CPU**, viene memorizzato l'indirizzo della **RAM** da cui inizia il blocco dati; questo indirizzo e' stato chiamato simbolicamente **BaseDati** (indirizzo di partenza del blocco dati) ed ha lo stesso significato del **BaseCodice**.
In un ulteriore registro della **CPU**, viene memorizzato l'indirizzo della **RAM** in cui si trova un eventuale dato a cui fa' riferimento l'istruzione da eseguire; questo indirizzo e' stato chiamato simbolicamente **OffsetDati** (spiazzamento all'interno del blocco dati) ed ha lo stesso significato di **OffsetCodice**.
Anche in questo caso quindi, mentre **BaseDati** e' un indirizzo fisso, **OffsetDati** puo' essere continuamente aggiornato attraverso le istruzioni del programma in esecuzione; in questo modo possiamo indicare alla **CPU** l'indirizzo del dato a cui vogliamo accedere.
Riprendendo il precedente esempio, consideriamo un **BaseDati** del programma in esecuzione, espresso in esadecimale dall'indirizzo **0DF0h**; tutto cio' significa che il blocco dati del programma inizia dalla cella n. **0DF0h** della **RAM**. Se la prossima istruzione da eseguire fa' riferimento ad un dato con **OffsetDati** pari a **00C8h**, allora l'indirizzo effettivo di questo dato e':

`0DF0h + 00C8h = 0EB8h`

Possiamo dire quindi che il prossimo dato da elaborare inizia dalla cella n. **0EB8h** della **RAM**; per poter accedere a questa cella, la **CPU** deve caricare l'indirizzo fisico **0EB8h** sull'**Address Bus**.
In definitiva, la somma:

`BaseDati + OffsetDati`

fornisce alla **CPU** l'indirizzo effettivo della cella della **RAM** da cui inizia il contenuto del prossimo dato da elaborare; possiamo paragonare **OffsetDati** all'indirizzo di una sorta di cursore che si muove all'interno del blocco dati del programma in esecuzione.

La tecnica di indirizzamento appena esposta viene definita **lineare** (in inglese si usa anche il termine **flat**); questo termine e' riferito al fatto che il programmatore vede la **RAM** come un vettore di celle, cioe' come una sequenza lineare di celle consecutive e contigue. Come abbiamo visto nel precedente capitolo, questo e' esattamente lo stesso punto di vista della **CPU**; si tratta quindi di una situazione ottimale, che permette al programmatore di accedere alla **RAM** nel modo piu' semplice e intuitivo. Ad ogni coppia **Base:Offset** specificata dal programmatore, corrisponde una e una sola cella fisicamente presente nella **RAM**; analogamente, ad ogni cella fisicamente presente nella **RAM**, corrisponde una e una sola coppia **Base:Offset**.
Appare evidente il fatto che per poter gestire un indirizzamento di tipo lineare, i registri interni della **CPU** devono avere una ampiezza in bit sufficiente per contenere un qualunque indirizzo di memoria; in sostanza, la **CPU** con **Address Bus** a **16** bit dell'esempio precedente, dovrebbe essere dotata di registri interni almeno a **16** bit. Nel caso generale, l'ampiezza in bit dei registri interni di una **CPU** rispecchia fedelmente l'architettura della **CPU** stessa; possiamo dire quindi che in linea di principio, una **CPU** con architettura (**Data Bus**) a **n** bit sara' dotata di registri interni anch'essi a **n** bit. Affinche' sia possibile l'indirizzamento di tipo lineare, anche l'**Address Bus** di questa **CPU** deve avere quindi una ampiezza massima di **n** bit; in una situazione di questo genere, i registri interni della **CPU** sono in grado di contenere l'indirizzo di una qualunque cella fisicamente presente nella **RAM**. In effetti, questo e' proprio cio' che accade in molte piattaforme hardware; un esempio pratico e' rappresentato dai computers **Apple Macintosh** equipaggiati con le **CPU** della **Motorola**.

9.1 La modalita' di indirizzamento reale 8086

La modalita' di indirizzamento lineare appena descritta, e' un vero e proprio paradiso per il programmatore; in questa modalita' infatti, il programmatore si trova a gestire nei propri programmi, indirizzi che fanno riferimento diretto a celle fisicamente presenti nel vettore della **RAM**.

Nel caso pero' della piattaforma hardware basata sulle **CPU** della famiglia **80x86**, la situazione e' purtroppo ben diversa da quella appena descritta; per rendercene conto, partiamo dal caso della **CPU 8080** che puo' essere definita come "l'origine di tutti i mali".

In un precedente capitolo abbiamo visto che l'**8080** e' dotata di **Address Bus** a **16** linee, **Data Bus** a **8** linee e quindi anche registri interni a **8** bit; la domanda che sorge spontanea e': come facciamo a gestire indirizzi fisici a **16** bit attraverso i registri a **8** bit?

Con un **Address Bus** a **16** linee l'**8080** puo' vedere al massimo $\mathbf{2^{16}=65536}$ byte di **RAM**; complessivamente, abbiamo quindi **65536** celle di memoria, i cui indirizzi sono compresi (in esadecimale) tra **0000h** e **FFFFh**. Per poter gestire nei nostri programmi questi indirizzi a **16** bit, abbiamo la necessita' di disporre di registri a **16** bit; l'**8080** ci mette pero' a disposizione registri interni a **8** bit, attraverso i quali possiamo rappresentare solo gli indirizzi compresi tra **00h** e **FFh**. Per risolvere questo problema, i progettisti della **Intel** hanno fatto in modo che i registri interni a **8** bit dell'**8080** possano essere usati anche in coppia; in questo modo, abbiamo a disposizione registri da **8+8** bit, attraverso i quali possiamo gestire indirizzi lineari a **16** bit.

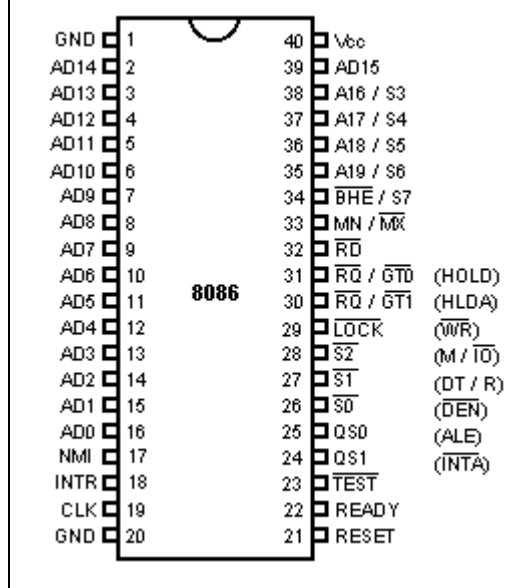
Riassumendo quindi, l'**8080** ha a disposizione una **RAM** da **65536** byte (**64** Kb); un programma da eseguire viene caricato in una porzione di questi **64** Kb, e grazie alle coppie di registri da **8+8** bit, puo' essere indirizzato attraverso la tecnica mostrata in Figura 1.

Nel caso dell'**8080** quindi, i progettisti della **Intel** hanno risolto il problema in modo relativamente semplice; bisogna anche tenere presente che quando comparve questa **CPU** (primi anni **70**), **64** Kb di **RAM** erano una enormita', e l'utilizzo di una architettura a **16** bit avrebbe comportato complicazioni circuitali eccessive e costi di produzione proibitivi.

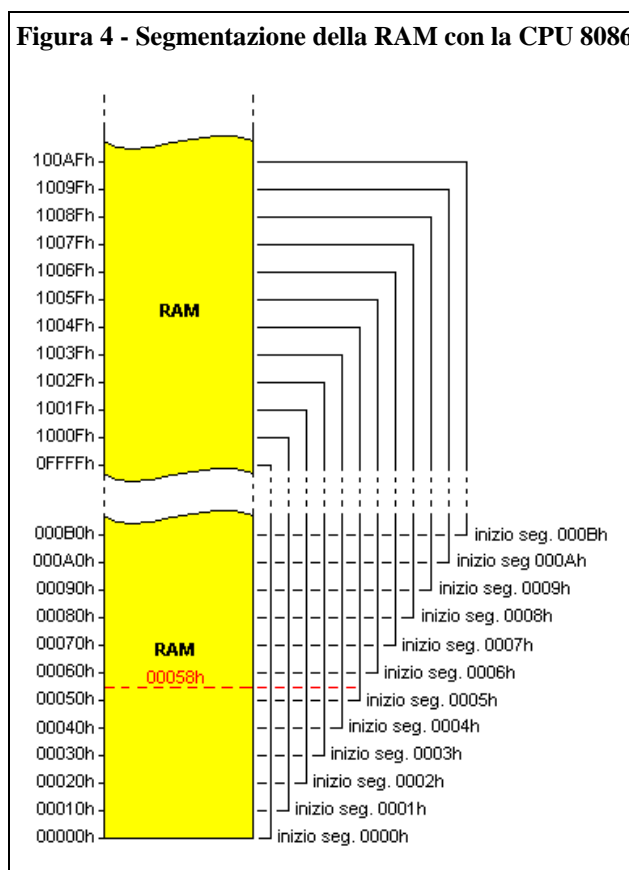
I problemi di indirizzamento dell'**8080** erano pero' ben poca cosa rispetto a quello che stava per succedere nel giro di pochi anni; nell'**Agosto** del **1981**, infatti, inizia ufficialmente l'era dei **Personal Computers** (**PC**). In quella data, la **IBM** mette in commercio un rivoluzionario computer chiamato **PC IBM XT** (la sigla **XT** sta per **eXtended Technology**); si tratta del primo computer destinato, non solo a centri di ricerca, universita', enti militari, etc, ma anche al mercato di fascia medio bassa rappresentato dalle piccole e medie aziende e da semplici utenti.

In poco tempo, l'**XT** diventa un vero e proprio standard nel mondo del computer, tanto che negli anni successivi, dopo memorabili battaglie giudiziarie, viene affiancato da una serie di cloni prodotti da numerose altre aziende; per rappresentare tutta questa famiglia di cloni dell'**XT**, viene coniata in quegli anni la famosa definizione di **PC IBM XT compatibili**.

La **CPU** che ha dominato il mondo degli **XT** compatibili e' stata sicuramente la **Intel 8086**; la Figura 3 mostra la piedinatura del chip che racchiude questa **CPU**.

Figura 3 - Piedinatura della CPU 8086

Osservando la Figura 3 balza subito agli occhi il fatto che l'**8086** e' predisposta per un **Address Bus** a **20** linee (da **AD0** a **AD15** e da **A16/S3** a **A19/S6**); con un **Address Bus** a **20** linee, l'**8086** puo' vedere sino a $\mathbf{2^{20}}$**=1048576** byte (**1** Mb) di memoria fisica. Questo aspetto suscito' all'epoca grande scalpore, tanto da far nascere un acceso dibattito tra gli esperti, che si interrogarono sulla effettiva utilita' di "cosi' tanta" memoria **RAM**!
L'**8086** e' una **CPU** con architettura a **16** bit, e dispone quindi di un **Data Bus** a **16** linee e di registri interni a **16** bit; osservando la Figura 3 si nota che per la connessione al **Data Bus** vengono sfruttate le prime **16** linee dell'**Address Bus** (da **AD0** a **AD15**). L'**Address Bus**, una volta che ha trasmesso alla **RAM** l'indirizzo della cella a cui dobbiamo accedere, ha praticamente esaurito il proprio lavoro; a questo punto la **CPU** puo' sfruttare i terminali da **AD0** a **AD15** per connettersi al **Data Bus**.
Per l'**8086** si pone lo stesso interrogativo dell'**8080**: come facciamo a gestire indirizzi fisici a **20** bit attraverso i registri a **16** bit?
In base a quanto e' stato detto in precedenza per l'**8080**, si potrebbe pensare che sull'**8086** sia possibile utilizzare in coppia i registri da **16** bit; i progettisti della **Intel** hanno escogitato invece una soluzione ben piu' contorta, che ha avuto colossali ripercussioni su tutta la famiglia delle **CPU 80x86**.
Osserviamo innanzi tutto che, come e' stato gia' detto, con un **Address Bus** a **20** linee l'**8086** puo' gestire $\mathbf{2^{20}}$**=1048576** byte (**1** Mb) di memoria fisica; complessivamente, abbiamo quindi **1048576** celle di memoria, i cui indirizzi sono compresi (in esadecimale) tra **00000h** e **FFFFFh**. Per poter gestire nei nostri programmi questi indirizzi a **20** bit, abbiamo la necessita' di disporre di registri a **20** bit; l'**8086** ci mette pero' a disposizione registri interni a **16** bit, attraverso i quali possiamo rappresentare solo gli indirizzi compresi tra **0000h** e **FFFFh**, per un totale di **65536** byte (**64** Kb) di **RAM**.
Per risolvere questo problema, i progettisti della **Intel** hanno deciso di suddividere **"idealmente"** la **RAM**, in tanti blocchi da **64** Kb ciascuno, chiamati **segmenti di memoria**; per indirizzare i vari segmenti di memoria, e' stato inoltre introdotto il concetto fondamentale di **indirizzo logico**. Un indirizzo logico, e' formato da una coppia indicata simbolicamente con **Seg:Offset**; il significato di questa coppia e' formalmente simile a quello delle coppie **Base:Offset**. La componente **Seg** e' un valore a **16** bit che indica il segmento di memoria a cui vogliamo accedere; con **16** bit possiamo specificare un qualunque segmento di memoria compreso tra **0000h** e **FFFFh**. La componente **Offset** e' un valore a **16** bit che indica uno spiazzamento all'interno del segmento di memoria a cui vogliamo accedere; con **16** bit possiamo specificare un qualunque spiazzamento compreso tra

0000h e **FFFFh**.
Seguendo allora un ragionamento logico, possiamo dedurre che i **1048576** byte di **RAM** indirizzabili dall'**8086** vengono suddivisi in:
`1048576 / 65536 =` 2^{20} `/` 2^{16} `=` 2^{20-16} `=` 2^4 `= 16` segmenti di memoria.
Volendo accedere allora alla trentesima cella del quarto segmento di memoria, dobbiamo specificare semplicemente l'indirizzo logico **3:29** (in esadecimale, **0003h:001Dh**); le deduzioni logiche appena esposte, sono pero' completamente sbagliate!
Alla **Intel** hanno notato che con la componente **Seg** a **16** bit e' possibile specificare teoricamente uno tra **65536** possibili segmenti di memoria (da **0000h** a **FFFFh**); analogamente, con la componente **Offset** a **16** bit e' possibile specificare uno tra **65536** possibili spiazzamenti (da **0000h** a **FFFFh**). In sostanza, attraverso le coppie **Seg:Offset** possiamo gestire **65536** segmenti di memoria, ciascuno dei quali e' formato da **65536** byte; complessivamente quindi, con questo sistema e' possibile indirizzare ben:
`65536 * 65536 =` 2^{16} `*` 2^{16} `=` 2^{16+16} `=` 2^{32} `= 4294967296 byte = 4 Gb` di memoria **RAM**!
Nei primi anni **80** pero' i Gb esistevano solo nei film di fantascienza; alla **Intel** si misero allora al lavoro per escogitare un sistema che consentisse di indirizzare con le coppie logiche **Seg:Offset**, unicamente **1** Mb di **RAM**. Alla fine, salto' fuori un'idea a dir poco cervellotica che, come e' stato gia' detto, ha avuto pesanti ripercussioni sulla architettura di tutte le **CPU** della famiglia **80x86**; **si raccomanda vivamente di leggere con molta attenzione cio' che viene esposto nel seguito del capitolo, perche' la padronanza di questi concetti e' di vitale importanza per chi vuole imparare a programmare seriamente un PC**.
In base all'idea scaturita dalle menti contorte dei progettisti della **Intel**, con le coppie **Seg:Offset** e' possibile rappresentare effettivamente **65536** segmenti di memoria differenti, ciascuno dei quali e' formato da **65536** byte; la novita' importante e' data pero' dal fatto che ciascun segmento di memoria, viene fatto partire da un distinto indirizzo fisico che deve essere un multiplo intero di **16** (cioe', **0**, **16**, **32**, **48**, **64**, etc). In questo modo si viene a creare la situazione illustrata in Figura 4; questa figura si riferisce in particolare ai primi **12** segmenti di memoria della **RAM**.

Figura 4 - Segmentazione della RAM con la CPU 8086

Nella terminologia usata dalla **Intel**, un blocco di memoria da **16** byte viene definito **paragrafo**; e' importante anche osservare che il valore **16**, in esadecimale si scrive **10h**, e che il valore **65535**, in esadecimale si scrive **FFFFh**. Tenendo conto di questi aspetti, possiamo notare in Figura 4 che:

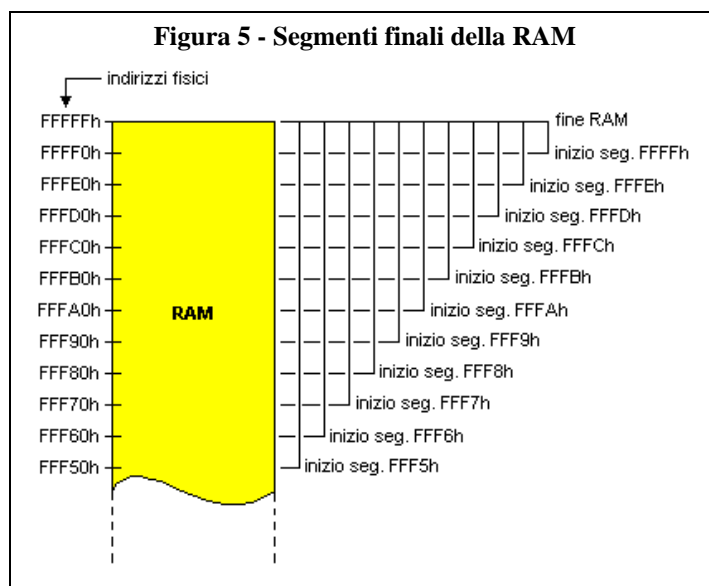
***** il segmento di memoria n. **0000h** parte dall'indirizzo fisico **00000h** e termina all'indirizzo fisico:
`00000h + 0FFFFh = 0FFFFh`
***** il segmento di memoria n. **0001h** parte dall'indirizzo fisico **00010h** e termina all'indirizzo fisico:
`00010h + 0FFFFh = 1000Fh`
***** il segmento di memoria n. **0002h** parte dall'indirizzo fisico **00020h** e termina all'indirizzo fisico:
`00020h + 0FFFFh = 1001Fh`
***** il segmento di memoria n. **0003h** parte dall'indirizzo fisico **00030h** e termina all'indirizzo fisico:
`00030h + 0FFFFh = 1002Fh`
e cosi' via.

Ciascun segmento di memoria parte quindi da un distinto indirizzo fisico multiplo intero di **16**; per esprimere questa situazione, si dice in gergo che i vari segmenti di memoria sono **paragraph aligned** (allineati al paragrafo).

Osserviamo ora che:
$1048576 / 16 = 2^{20} / 2^{4} = 2^{20-4} = 2^{16} = 65536$
I **1048576** byte della **RAM** contengono quindi **65536** paragrafi, ciascuno dei quali segna l'inizio di un nuovo segmento di memoria; cio' conferma che con questo sistema e' possibile avere fisicamente **65536** segmenti di memoria differenti. Da queste considerazioni si deduce quindi che l'ultimo segmento di memoria e' il n. **65535** (cioe', il n. **FFFFh**), e che il suo indirizzo fisico iniziale e' **FFFF0h**.
Lo schema descritto in Figura 4 presenta il pregio di permettere alla **CPU** di convertire con estrema facilita' un indirizzo logico **Seg:Offset** in un indirizzo fisico a **20** bit; per capire come avviene questa conversione, e' sufficiente osservare in Figura 4 che un generico segmento di memoria n. **XXXXh** parte dall'indirizzo fisico **XXXX0h**. Questo indirizzo fisico si ottiene quindi moltiplicando **XXXXh** per **10h** (cioe' per **16**); al risultato **XXXX0h** bisogna poi sommare la componente **Offset** e il gioco e' fatto. Dato quindi l'indirizzo logico **Seg:Offset**, il corrispondente indirizzo fisico a **20** bit si ricava dalla formula:
`(Seg * 16) + Offset`
Supponiamo ad esempio di avere l'indirizzo logico **0C2Fh:AB32h**; questo indirizzo logico rappresenta la cella n. **AB32h** del segmento di memoria n. **0C2Fh**. Il corrispondente indirizzo fisico a **20** bit e' dato allora da:
`(0C2Fh * 10h) + AB32h = 0C2F0h + AB32h = 16E22h`
Moltiplicare un numero esadecimale per **16**, cioe' per 16^{1}, equivale come sappiamo a far scorrere di un posto verso sinistra tutte le sue cifre; il posto rimasto libero a destra viene riempito con uno **0**. Analogamente, tenendo presente che la **CPU** lavora con il codice binario, moltiplicare un numero binario per **16**, cioe' per 2^{4}, equivale come sappiamo a far scorrere di quattro posti verso sinistra tutte le sue cifre; i quattro posti rimasti liberi a destra vengono riempiti con quattro **0**. In definitiva, la conversione di un indirizzo logico **Seg:Offset** espresso in binario, in un indirizzo fisico a **20** bit, comporta uno **shift** di **4** posti verso sinistra delle cifre di **Seg**, e una successiva somma con **Offset**; come vedremo nel prossimo capitolo, la **CPU** dispone di un apposito circuito, che converte ad altissima velocita', un indirizzo logico **Seg:Offset** in un indirizzo fisico a **20** bit.
Lo schema di segmentazione della **RAM** illustrato in Figura 4 presenta anche alcuni inconvenienti; un evidente inconveniente e' rappresentato dal fatto che, a differenza di quanto accade con lo schema di indirizzamento di tipo lineare, l'indirizzamento di tipo logico non garantisce piu' la corrispondenza biunivoca tra indirizzi logici **Seg:Offset** e indirizzi fisici della **RAM**. Nel precedente esempio, abbiamo visto che l'indirizzo logico **0C2Fh:AB32h** e' associato univocamente all'indirizzo fisico **16E22h**; viceversa, l'indirizzo fisico **16E22h** puo' essere rappresentato con differenti coppie **Seg:Offset**. Per rendercene conto, possiamo osservare in Figura 4 che questa

situazione, e' legata al fatto che i vari segmenti di memoria sono parzialmente sovrapposti tra loro; ne consegue che, ad esempio, l'indirizzo fisico **00058h** (colorato in rosso in Figura 4), e' rappresentabile con ben **6** coppie **Seg:Offset** differenti, e cioe':

0005h:0008h
0004h:0018h
0003h:0028h
0002h:0038h
0001h:0048h
0000h:0058h
Nota importante.
Nel caso in cui il programmatore abbia la necessita' di convertire un indirizzo fisico a **20** bit in uno dei corrispondenti indirizzi logici, e' possibile ricorrere ad un metodo molto semplice; prima di tutto si prende l'indirizzo fisico a **20** bit, lo si converte per comodita' in esadecimale a **5** cifre, e lo si suddivide in due gruppi di cifre. Il primo gruppo e' formato unicamente dalla cifra meno significativa e rappresenta la componente **Offset**; il secondo gruppo e' formato dalle restanti **4** cifre piu' significative e rappresenta la componente **Seg**.
Come verifica, consideriamo l'indirizzo fisico a **20** bit **AFC8Dh**; applicando il metodo appena descritto, si ottiene l'indirizzo logico **AFC8h:Dh**, cioe' **AFC8h:000Dh**. Si puo' facilmente constatare che:
(AFC8h * 10h) + 000Dh = AFC80h + 000Dh = AFC8Dh
Questa tecnica permette di ottenere la coppia logica **Seg:Offset** caratterizzata da una componente **Seg** che e' la piu' grande possibile, e da una componente **Offset** che e' la piu' piccola possibile; come vedremo in un capitolo successivo, un indirizzo logico di questo genere viene chiamato **indirizzo logico normalizzato**. Attraverso l'esempio appena illustrato, si puo' constatare che in un indirizzo logico normalizzato come **AFC8h:000Dh**, la componente **Seg** (**AFC8h**) e' associata all'indirizzo fisico multiplo di **16** (**AFC80h**) che piu' si avvicina per difetto all'indirizzo fisico **AFC8Dh**; questa proprieta' e' molto importante, e verra' ampiamente sfruttata in questo capitolo e nei capitoli successivi.
Un altro serio inconveniente dello schema di segmentazione della **RAM** illustrato in Figura 4, e' rappresentato dal fatto che, come molti avranno intuito, gli ultimi segmenti di memoria sono incompleti; questi segmenti hanno cioe' una dimensione inferiore a **65536** byte. Per rendercene conto, osserviamo la Figura 5 che mostra gli ultimi **11** segmenti di memoria della **RAM**.

Figura 5 - Segmenti finali della RAM

Per capire bene questo problema, bisogna ricordare che la **8086**, a causa dell'**Address Bus** a **20** linee, e' in grado di vedere solo **100000h** byte di memoria (da **00000h** a **FFFFFh**); come vedremo in seguito, questa limitazione vale per qualsiasi **80x86** in emulazione **8086** (anche in presenza di piu' di **1** Mb di **RAM**).

Cominciamo dall'ultimo segmento di memoria, cioe' dal segmento n. **FFFFh**; all'interno di questo segmento possiamo accedere ai vari byte attraverso gli indirizzi logici **FFFFh:0000h**, **FFFFh:0001h**, **FFFFh:0002h** e cosi' via, sino ad arrivare a **FFFFh:000Fh**. Giunti a questo punto, possiamo constatare che l'indirizzo logico **FFFFh:000Fh** corrisponde all'indirizzo fisico:
`(FFFFh * 10h) + 000Fh = FFFF0h + 000Fh = FFFFFh`
Ma **FFFFFh** non e' altro che l'indirizzo fisico dell'ultima cella della **RAM**; tutto cio' significa che l'ultimo segmento di memoria, cioe' il n. **FFFFh**, e' formato da appena **16** byte (**10h** byte)!
Passiamo ora al penultimo segmento di memoria, cioe' al segmento n. **FFFEh**; all'interno di questo segmento possiamo accedere ai vari byte attraverso gli indirizzi logici **FFFEh:0000h**, **FFFEh:0001h**, **FFFEh:0002h** e cosi' via, sino ad arrivare a **FFFEh:001Fh**. Giunti a questo punto, possiamo constatare che l'indirizzo logico **FFFEh:001Fh** corrisponde all'indirizzo fisico:
`(FFFEh * 10h) + 001Fh = FFFE0h + 001Fh = FFFFFh`
Ma **FFFFFh** non e' altro che l'indirizzo fisico dell'ultima cella della **RAM**; tutto cio' significa che il penultimo segmento di memoria, cioe' il n. **FFFEh**, e' formato da appena **32** byte (**20h** byte)!
A questo punto la situazione appare chiara, per cui si puo' facilmente intuire che, il terzultimo segmento di memoria (**FFFDh**) e' formato da soli **48** byte (**30h** byte), il quartultimo segmento di memoria (**FFFCh**) e' formato da soli **64** byte (**40h** byte), il quintultimo segmento di memoria (**FFFBh**) e' formato da soli **80** byte (**50h** byte) e cosi' via; quanti sono in totale i segmenti di memoria incompleti?
Per rispondere a questa domanda basta osservare che:
$65536 / 16 = 2^{16} / 2^{4} = 2^{16-4} = 2^{12} = 4096$
In un segmento di memoria completo ci sono quindi **4096** paragrafi; possiamo dire allora che degli ultimi **4096** segmenti di memoria, solo il primo di essi (il n. **F000h**) e' completo. Tutti gli altri **4095** sono incompleti; il n. **F001h** e' piu' corto di **16** byte, il n. **F002h** e' piu' corto di **32** byte, il n. **F003h** e' piu' corto di **48** byte e cosi' via, sino al n. **FFFFh** che risulta piu' corto di ben **65520** byte.

Cosa succede se, con una **CPU 8086**, proviamo ad accedere ad un indirizzo logico come **FFFFh:0010h** che corrisponde ad un indirizzo fisico inesistente?
La **CPU** trova questo indirizzo logico e lo converte nell'indirizzo fisico:
`(FFFFh * 10h) + 10h = FFFF0h + 10h = 100000h`
Tradotto in binario questo indirizzo fisico e' espresso da **100000000000000000000b**; come si puo' notare, si tratta di un indirizzo formato da **21** bit. L'**Address Bus** dell'**8086** e' formato da sole **20** linee, per cui non puo' contenere un indirizzo a **21** bit; quello che succede e' che la cifra piu' significativa (**1**) dell'indirizzo a **21** bit viene persa e si ottiene l'indirizzo a **20** bit **00000000000000000000b**!
In pratica, se proviamo ad accedere all'indirizzo logico **FFFFh:0010h**, ci ritroviamo all'indirizzo fisico **00000h**; analogamente, ripetendo il calcolo precedente si puo' constatare che, se proviamo ad accedere all'indirizzo logico **FFFFh:0011h**, ci ritroviamo all'indirizzo fisico **00001h**, se proviamo ad accedere all'indirizzo logico **FFFFh:0012h**, ci ritroviamo all'indirizzo fisico **00002h**, se proviamo ad accedere all'indirizzo logico **FFFFh:0013h**, ci ritroviamo all'indirizzo fisico **00003h** e cosi' via.
Il fenomeno appena descritto e' caratteristico della **CPU 8086** e prende il nome di **wrap around** (attorcigliamento); il programmatore deve evitare accuratamente di incappare in questa situazione che spesso porta un programma ad invadere aree della **RAM** riservate al sistema operativo, con conseguente crash del computer!

Tutti i concetti fondamentali precedentemente esposti, rappresentano la cosiddetta **modalita' di indirizzamento reale 8086** (o semplicemente **modalita' reale**); il termine **reale** si riferisce al fatto che una qualunque coppia logica valida **Seg:Offset**, si riferisce ad un indirizzo realmente esistente all'interno del Mb di **RAM** visibile dall'**8086**. Il programmatore che intende scrivere programmi per questa modalita' operativa delle **CPU 80x86**, deve avere una padronanza totale di questi concetti; infatti, nello scrivere un programma destinato alla modalita' reale, il programmatore si trova continuamente alle prese con indirizzamenti che devono essere espressi sotto forma di coppie logiche **Seg:Offset**, che verranno poi tradotte dalla **CPU** in indirizzi fisici a **20** bit da caricare sull'**Address Bus**.
Come vedremo nei capitoli successivi, per consentire al programmatore di gestire al meglio questa situazione, la **CPU 8086** mette a disposizione una serie di registri a **16** bit; in particolare, sono presenti appositi registri destinati a contenere indirizzi logici del tipo **Seg:Offset**. Un qualunque registro destinato a contenere una componente **Seg** viene chiamato **Segment Register** (registro di segmento); un qualunque registro destinato a contenere una componente **Offset** viene chiamato **Pointer Register** (registro puntatore). Utilizzando questi registri e' possibile organizzare un programma nel modo piu' adatto alle nostre esigenze; nel caso piu' semplice, per la corretta gestione di un programma in esecuzione vengono utilizzati: un registro di segmento per il blocco codice, un registro di segmento per il blocco dati, un registro puntatore per il blocco codice e uno o piu' registri puntatori per il blocco dati.
Apparentemente la situazione sembra molto simile al caso dell'indirizzamento lineare illustrato in Figura 1; in realta', tra una coppia **Base:Offset** e una coppia **Seg:Offset** esistono notevoli differenze sostanziali. La componente **Base** di un indirizzo lineare e' l'indirizzo fisico iniziale di un blocco di programma; la componente **Seg** di un indirizzo logico individua invece uno tra i possibili **65536** segmenti di memoria allineati al paragrafo all'interno dell'unico Mb di **RAM** gestibile dall'**8086**. La componente **Offset** di un indirizzo lineare e' un qualunque spiazzamento calcolato rispetto all'indirizzo fisico **Base**; la componente **Offset** di un indirizzo logico e' uno spiazzamento compreso tra **0000h** e **FFFFh**, calcolato rispetto all'indirizzo fisico a **20** bit da cui inizia il segmento di memoria **Seg**.

E' importante ribadire ancora una volta che un programmatore deve acquisire in modo completo i concetti appena esposti; in caso contrario non sara' mai in grado di scrivere un programma serio, ne' con l'**Assembly**, ne' con i linguaggi di alto livello. L'importanza enorme di questi concetti e' legata in particolare al fatto che, come vedremo tra breve, la modalita' reale **8086** continua ad esistere anche su tutte le **CPU** della famiglia **80x86**; questo discorso vale quindi anche per le **CPU** di classe **Pentium I**, **II**, **III**, etc.

9.2 La modalita' di indirizzamento protetto

Nel **1982** la **Intel** mette in commercio una nuova **CPU**, chiamata **80186**; nel mondo dei **PC** questa **CPU** passa praticamente inosservata a causa del fatto che si tratta semplicemente di una versione piu' sofisticata dell'**8086**. La vera novita' arriva invece nello stesso anno con l'inizio della produzione della **CPU 80286**; questa **CPU** e' dotata, come l'**8086**, di una architettura a **16** bit (**Data Bus** a **16** linee e registri interni a **16** bit), ma presenta un **Address Bus** formato da ben **24** linee!
Con un **Address Bus** a **24** linee l'**80286** puo' vedere sino a 2^{24}**=16777216** byte (**16** Mb) di **RAM** fisica; per poter accedere a questa "gigantesca" quantita' di **RAM**, l'**80286** fornisce, per la prima volta, una nuova modalita' operativa chiamata **modalita' di indirizzamento protetto**.
Sostanzialmente, questa modalita' operativa permette al programmatore di accedere alla **RAM** attraverso lo schema illustrato in Figura 1; questo schema prevede, come sappiamo, l'utilizzo degli indirizzi **Base:Offset** di tipo lineare. A tale proposito, i registri di segmento a **16** bit dell'**80286**, vengono utilizzati per indicizzare una tabella che si trova in memoria, e che contiene la descrizione completa dei vari blocchi che formano il programma in esecuzione; questa descrizione comprende,

in particolare, la componente **Base** a **24** bit che esprime, come sappiamo, l'indirizzo fisico da cui inizia un determinato blocco di programma.
Tra i programmatori pero' si diffonde subito una certa delusione, dovuta al fatto che l'architettura a **16** bit dell'**80286**, permette di esprimere indirizzi **Base:Offset** con la componente **Offset** a soli **16** bit; come accade quindi per l'**8086**, anche con l'**80286** la componente **Offset** puo' assumere solamente i valori compresi tra **0000h** e **FFFFh**. In base a queste considerazioni, possiamo dire che con l'**80286** il programmatore ha la possibilita' di accedere in modalita' protetta a ben **16** Mb di memoria fisica, attraverso indirizzi **Base:Offset** dotati di componente **Base** a **24** bit; a causa pero' dell'architettura a **16** bit di questa **CPU**, la componente **Offset** rimane a **16** bit, per cui la memoria risulta ancora suddivisa in blocchi che non possono superare la dimensione di **64** Kb.
In ogni caso, l'**80286** introduce sicuramente numerose e importanti novita' che aprono tra l'altro la strada ai sistemi operativi, come **Windows**, capaci di eseguire piu' programmi contemporaneamente (sistemi operativi **multitasking**); a tale proposito, l'**80286** fornisce anche il supporto necessario ai sistemi operativi, per evitare che due o piu' programmi contemporaneamente in esecuzione, possano danneggiarsi a vicenda. Questi meccanismi di protezione originano appunto la definizione di **modalita' protetta**; un programma scritto invece per la modalita' reale, ha la possibilita', ad esempio, di invadere liberamente le aree di memoria riservate al sistema operativo, con il rischio elevato di provocare un crash del computer.

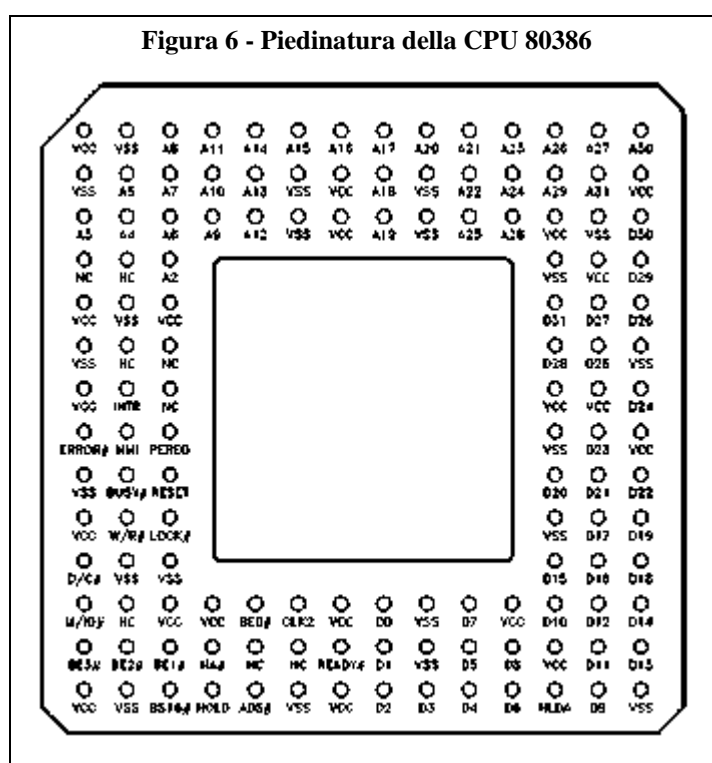
Prima di mettere in commercio l'**80286**, la **Intel** si trova ad affrontare una situazione molto delicata, dovuta al fatto che nel frattempo, il mondo dei **PC** e' stato letteralmente invaso da una enorme quantita' di software scritto esplicitamente per la modalita' reale delle **CPU 8086**; con l'arrivo dell'**80286**, c'e' quindi il rischio che tutto questo software diventi inutilizzabile. Per evitare questa eventualita', la **Intel** progetta l'**80286** rendendo questa **CPU** compatibile con l'**8086**; in particolare, l'**80286** conserva la stessa struttura interna dei registri dell'**8086**, garantendo cosi' che tutti i programmi scritti per la modalita' reale dell'**8086**, possano girare senza nessuna modifica sull'**80286**.
In piu', bisogna anche aggiungere che non appena si accende un computer equipaggiato con una **80286**, la **CPU** viene inizializzata in modalita' di indirizzamento reale **8086**; in questa modalita' operativa, l'**80286** si comporta come se fosse una **8086**.
Ben presto pero' i progettisti della **Intel** scoprono un ulteriore problema piuttosto curioso, che si verifica quando l'**80286** si trova in modalita' reale; questo problema e' legato al fatto che il **wrap around** caratteristico dell'**8086**, non si verifica piu' con l'**80286**. Per capire il perche' riprendiamo un precedente esempio nel quale abbiamo visto che l'indirizzo logico **FFFFh:0010h** corrisponde all'indirizzo fisico **100000h**, che necessita di almeno **21** bit; questo indirizzo provoca naturalmente il **wrap around** sull'**Address Bus** a **20** linee dell'**8086**, mentre con l'**80286**, grazie all'**Address Bus** a **24** linee, e' un indirizzo perfettamente valido!
Non ci vuole molto a capire che con la **CPU 80286**, pur lavorando in modalita' reale, e' possibile indirizzare anche una certa quantita' di memoria che si trova subito dopo l'unico Mb di **RAM** visibile dall'**8086**; con l'**8086**, il limite massimo e' rappresentato dall'indirizzo fisico **FFFFFh**, a cui possiamo associare, ad esempio, l'indirizzo logico **FFFFh:000Fh**. Con l'**80286**, possiamo scavalcare questo limite indirizzando, sempre in modalita' reale, tutta la memoria compresa tra gli indirizzi logici **FFFFh:0010h** e **FFFFh:FFFFh**; questi indirizzi logici corrispondono agli indirizzi fisici **100000h** e **10FFEFh**, per cui, l'ulteriore memoria indirizzabile con l'**80286** e' pari a:

```
10FFEFh - 100000h + 1h = FFF0h = 65520 byte
```

Il **+1** tiene conto del byte iniziale che si trova all'indirizzo **100000h**; ricordiamoci a tale proposito dell'errore **fuori di uno** citato nel precedente capitolo.
Questo blocco da **65520** byte e' stato chiamato **High Memory Area** (area di memoria alta), e puo' esistere quindi solo in presenza di una **CPU 80286** o superiore; come si puo' notare, si tratta di un blocco che ha appena **16** byte (un paragrafo) in meno rispetto ad un segmento di memoria da **65536** byte.
Per evitare qualunque problema di compatibilita' con l'**8086**, la **Intel** ha fatto in modo che

sull'**80286** sia possibile disabilitare la linea **A20** dell'**Address Bus**; in questo modo, l'**80286** quando lavora in modalita' reale, si ritrova con un **Address Bus** a **20** linee, attraverso il quale e' possibile indirizzare solo **1** Mb di **RAM** come accade con l'**8086**.
Riassumendo, possiamo dire quindi che l'**80286** e' capace di funzionare, sia in modalita' reale **8086**, sia in modalita' protetta; in modalita' reale gli indirizzamenti si svolgono con le coppie logiche **Seg:Offset** da **16+16** bit, mentre in modalita' protetta gli indirizzamenti si svolgono con le coppie **Base:Offset** da **24+16** bit.
L'**80286** e' stata subito utilizzata dalla **IBM** sui propri **PC**, determinando in questo modo la nascita di un nuovo standard chiamato **PC IBM AT** (la sigla **AT** sta per **Advanced Technology**); anche in questo caso il **PC IBM AT** e' stato subito affiancato da una numerosa serie di cloni, chiamati **PC IBM AT compatibili**.

La vera svolta nella storia della **Intel** arriva nel **1985** con la progettazione della nuova **CPU 80386**; la Figura 6 mostra la piedinatura (vista dal basso) del chip che racchiude questa **CPU**.

Figura 6 - Piedinatura della CPU 80386

Osserviamo subito che l'**80386** e' predisposta per un **Address Bus** formato da ben **32** linee (da **A0** a **A31**); con un **Address Bus** a **32** linee, l'**80386** e' in grado di indirizzare sino a **2^{32}=4294967296** byte (**4** Gb) di memoria **RAM** fisica!
Come accade per l'**80286**, anche l'**80386** rende disponibile la modalita' operativa protetta, che permette di accedere in modo lineare a tutta questa memoria; la novita' importante pero' e' rappresentata dal fatto che, come si puo' notare in Figura 6, l'**80386** e' una **CPU** con architettura a **32** bit, dotata quindi di **Data Bus** a **32** linee (da **D0** a **D31**) e numerosi registri a **32** bit. Per la precisione, i registri di segmento dell'**80386** continuano ad essere tutti a **16** bit; come al solito, questi registri vengono utilizzati in modalita' protetta per indicizzare una tabella che si trova in memoria e che contiene la descrizione completa dei vari blocchi che formano il programma in esecuzione. Questa descrizione comprende, in particolare, la componente **Base** a **32** bit che rappresenta l'indirizzo fisico da cui parte in memoria un determinato blocco di programma; l'aspetto fondamentale e' rappresentato pero' dal fatto che, questa volta, nelle coppie **Base:Offset** anche la componente **Offset** e' a **32** bit!
Un offset a **32** bit puo' spaziare da **00000000h** a **FFFFFFFFh**, permettendoci di accedere

linearmente a tutta la **RAM** disponibile; tutto cio' significa in sostanza che con la modalita' protetta dell'**80386**, sparisce finalmente la segmentazione a **64** Kb della memoria!

Nel progettare l'**80386** la **Intel** garantisce la piena compatibilita' con il software scritto per la modalita' reale **8086** e per la modalita' protetta a **24** bit delle **CPU 80286**; questa scelta comporta per l'**80386** una architettura interna piuttosto complessa. Come e' stato gia' detto, i registri di segmento dell'**80386** continuano ad essere a **16** bit, e in modalita' reale vengono utilizzati, naturalmente, per contenere la componente **Seg** di una coppia logica **Seg:Offset**; gli altri registri a **32** bit (compresi i registri puntatori), vengono ottenuti aggiungendo ulteriori **16** bit ai registri a **16** bit dell'**8086**.
Come accade per l'**80286**, non appena si accende un computer equipaggiato con una **80386**, la **CPU** viene inizializzata in modalita' reale con conseguente disabilitazione della linea **A20** dell'**Address Bus**; in queste condizioni, l'**80386** si comporta come se fosse una **8086** ed e' in grado di indirizzare direttamente **1** Mb di **RAM**.
In definitiva, l'**80386** e' una **CPU** capace di funzionare, sia in modalita' reale **8086**, sia in modalita' protetta; in modalita' reale gli indirizzamenti si svolgono con le coppie logiche **Seg:Offset** da **16+16** bit, mentre in modalita' protetta gli indirizzamenti si svolgono con le coppie **Base:Offset** da **32+32** bit.

Tutte le **CPU** successive all'**80386**, come l'**80486**, l'**80586** (classe **Pentium I**), l'**80686** (classe **Pentium II**), etc, si possono considerare come evoluzioni sempre piu' potenti e sofisticate della stessa **80386**; queste **CPU** formano una famiglia chiamata **80x86** e garantiscono pienamente la compatibilita' verso il basso. Questo significa che un programma scritto per una **8086**, gira senza nessuna modifica su una **CPU** di classe superiore; viceversa, un programma scritto esplicitamente per una **80586**, non puo' girare sulle **CPU 80486** o inferiori.
E' importante ribadire che tutte le **CPU** della famiglia **80x86**, all'accensione del computer, vengono inizializzate in modalita' reale **8086** con conseguente disabilitazione della linea **A20**; per queste **CPU** valgono quindi le stesse considerazioni gia' esposte per l'**80286** e per l'**80386**.

Un aspetto piuttosto interessante e' rappresentato dal fatto che una **CPU 80386** o superiore che lavora in modalita' reale, mette a disposizione molte sue caratteristiche architetturali senza la necessita' di passare in modalita' protetta; avendo a disposizione ad esempio una **CPU 80386** con architettura a **32** bit, e' possibile sfruttare interamente i registri a **32** bit senza uscire dalla modalita' reale. Tutto cio' si rivela particolarmente utile nel caso di operazioni che coinvolgono numeri a **32** bit; sfruttando i registri a **32** bit dell'**80386**, possiamo effettuare queste operazioni in modo rapidissimo.
Le considerazioni appena esposte creano un dubbio relativo all'uso dei registri puntatori a **32** bit negli indirizzamenti in modalita' reale; abbiamo visto che in questa modalita' operativa, vengono utilizzati indirizzi logici del tipo **Seg:Offset** con le due componenti **Seg** e **Offset** che hanno entrambe una ampiezza di **16** bit. Nel caso della componente **Seg** non ci sono problemi in quanto le **CPU 80386** e superiori continuano ad avere registri di segmento a **16** bit; e' possibile utilizzare invece un registro puntatore a **32** bit per contenere una componente **Offset** a **32** bit?
La risposta e' affermativa in quanto le **CPU** come l'**80386** permettono al programmatore di selezionare, anche in modalita' reale, l'ampiezza in bit della componente **Offset** di un indirizzo logico **Seg:Offset**; la tecnica da utilizzare coinvolge aspetti che richiedono la conoscenza della programmazione in modalita' protetta delle **CPU 80x86**. A sua volta, la programmazione in modalita' protetta richiede una solidissima conoscenza del linguaggio **Assembly**, per cui questo argomento verra' trattato in dettaglio nella apposita sezione **Modalita' Protetta** di questo sito; tutte le considerazioni svolte nella sezione **Assembly Base**, si riferiscono invece alla modalita' operativa reale delle **CPU 80x86** con indirizzamenti logici di tipo **Seg:Offset** da **16+16** bit.

9.3 Sistemi operativi per la modalita' reale: il DOS

Come e' stato detto in un precedente capitolo, il sistema operativo (**SO**) di un computer puo' essere paragonato al cruscotto di una automobile; sarebbe piuttosto difficile guidare una automobile senza avere a disposizione il volante, il freno, l'acceleratore, la spia dell'olio, etc. Attraverso questi strumenti l'automobilista puo' gestire facilmente una automobile senza la necessita' di conoscere il funzionamento interno del motore e dei vari meccanismi; un discorso analogo puo' essere fatto anche per il computer. Sarebbe praticamente impossibile infatti, usare un computer senza avere a disposizione un **SO**; attraverso il **SO** anche un utente privo di qualsiasi conoscenza di informatica puo' utilizzare il computer per usufruire di tutta una serie di servizi che permettono ad esempio di ascoltare musica, di giocare, di navigare in **Internet**, etc.
Nell'ambiente operativo rappresentato dalla modalita' reale delle **CPU 80x86**, il **SO** dominante (nel bene e nel male) e' stato ed e' sicuramente il **DOS** (**Disk Operating System**); questo **SO** venne scelto per la prima volta dalla **IBM** per equipaggiare il **PC IBM XT**. L'antenato del **DOS** e' il **CP/M** della **Digital**, destinato alla gestione dei computers con architettura a **8** bit; il **DOS** puo' essere definito come una evoluzione del **CP/M**, ed e' rivolto esplicitamente alla modalita' reale delle **CPU 80x86**. Per la verita' bisogna dire che il **DOS** e' un vero e proprio clone del **CP/M**; le cronache dell'epoca infatti riferiscono di un dipendente infedele della **Digital** che trafugo' il **CP/M** e lo vendette a due personaggi, **Bill Gates** e **Paul Allen**, che pochi anni prima avevano fondato una piccola software house chiamata inizialmente **Micro-Soft** (con il trattino). In quegli anni non esisteva nessuna legge sulla pirateria del software, per cui **Gates** e **Allen** la fecero franca; questo episodio emblematico segno' l'esordio di quello che sarebbe poi diventato il tanto discusso colosso **Microsoft**.

Il **DOS** e' un **SO** con interfaccia testuale (interfaccia a caratteri o alfanumerica); cio' significa che quando l'utente accende un computer gestito dal **DOS**, dopo le varie inizializzazioni si trova davanti ad uno schermo in modalita' testo, che mostra un cursore lampeggiante chiamato **Prompt del DOS**.
A questo punto l'utente, attraverso la tastiera, puo' impartire al **SO** i comandi desiderati; ciascun comando viene letto da un apposito programma chiamato **interprete dei comandi**, che dopo averne verificato la validita' provvede ad eseguirlo.

Ridotto all'essenziale il **DOS** e' formato da tre programmi contenuti nei tre file **IO.SYS**, **MSDOS.SYS** e **COMMAND.COM**; nella versione **IBM** del **DOS**, il file **IO.SYS** viene chiamato **IBMBIO.SYS**, mentre il file **MSDOS.SYS** viene chiamato **IBMDOS.SYS**.

Il programma **IO.SYS** si occupa innanzi tutto di un compito fondamentale che consiste nel caricamento in memoria e nella conseguente inizializzazione del **DOS**; si tratta quindi di un modulo che sfrutta ampiamente gli strumenti messi a disposizione dal **BIOS**.
Proprio attraverso il **BIOS**, il modulo **IO.SYS** fornisce anche una serie di procedure che si occupano della gestione a basso livello dell'hardware del computer; attraverso questo programma l'utente puo' richiedere i servizi a basso livello offerti da svariate periferiche. La gestione a basso livello delle periferiche viene resa possibile dai cosiddetti **device drivers** (piloti di dispositivo); si tratta di appositi programmi sviluppati dai produttori delle periferiche, che permettono al **DOS** di dialogare con tastiere, stampanti, mouse, etc, senza conoscerne il funzionamento interno.

Il programma **MSDOS.SYS** fornisce invece una numerosa serie di servizi che permettono ai programmi **DOS** di interfacciarsi ad alto livello con le periferiche del computer; grazie a questi servizi, un programma **DOS** puo' ad esempio inviare dati ad una generica stampante senza preoccuparsi degli aspetti legati al funzionamento a basso livello di questa periferica. Un altro importantissimo compito svolto da **MSDOS.SYS** consiste nella gestione del cosiddetto **file system** del computer; si tratta di un insieme di servizi di alto livello che permettono ai programmi **DOS** di

accedere in lettura e in scrittura alle memorie di massa (hard disk, floppy disk, etc), senza preoccuparsi dei dettagli relativi al metodo utilizzato per memorizzare fisicamente i dati su questi supporti.
Nel loro insieme, **IO.SYS** e **MSDOS.SYS** formano il cosiddetto **kernel** del **DOS**; il termine **kernel** deriva dal tedesco e significa **nucleo centrale**.

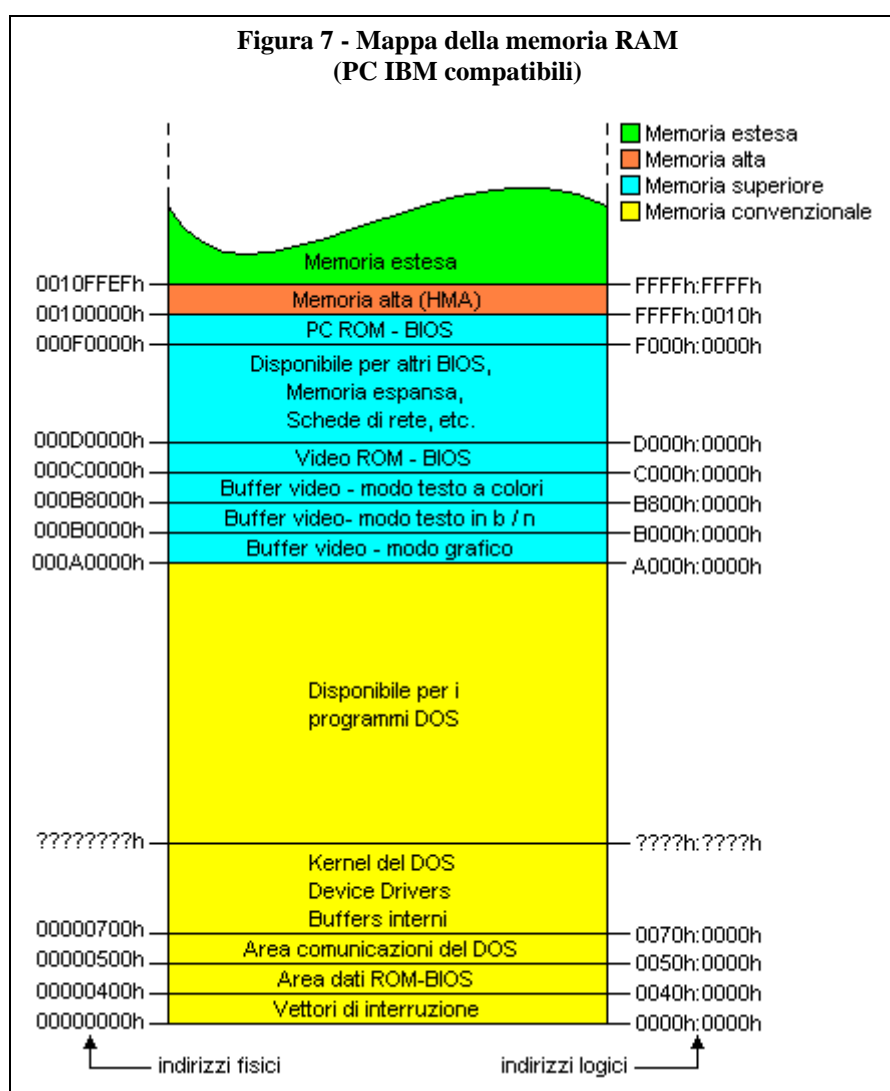
Il programma **COMMAND.COM** rappresenta l'interprete dei comandi del **DOS**; questo programma attiva un ciclo infinito in attesa che l'utente impartisca un comando. Quando cio' accade, **COMMAND.COM** verifica il comando, e se lo ritiene valido lo esegue tramite gli opportuni servizi offerti da **MSDOS.SYS** e **IO.SYS**. Se il comando non e' valido, **COMMAND.COM** produce un messaggio di errore del tipo:

```
Comando o nome file non valido
```

Se il **DOS** e' stato installato nella partizione **C:** del disco rigido, allora i tre file **IO.SYS**, **MSDOS.SYS** e **COMMAND.COM** si trovano nella cartella **C:**; trattandosi di file nascosti, per visualizzarli bisogna posizionarsi in **C:** e impartire il comando:

```
dir /ah
```

Per prendere confidenza con l'ambiente operativo offerto dal **DOS**, esaminiamo ora l'organizzazione della **RAM** del computer sotto questo **SO**; la Figura 7 illustra la mappa della **RAM**, relativa ad un generico **PC IBM** compatibile.

Figura 7 - Mappa della memoria RAM
(PC IBM compatibili)

Per trattare il caso piu' generale possibile supponiamo di avere a che fare con un **PC** dotato di **CPU 80386** o superiore e di piu' di **1** Mb di **RAM**; in questo modo possiamo anche osservare il punto di vista del **DOS** in relazione alla **RAM** che si trova oltre il primo **Mb**.
In Figura 7, sulla parte sinistra della **RAM** possiamo notare i vari indirizzi fisici a **32** bit; ovviamente questi indirizzi possono essere visti dall'**80386** solo in modalita' protetta. Sulla parte destra della **RAM** notiamo invece i vari indirizzi logici di tipo **Seg:Offset** a **16+16** bit cosi' come vengono visti dal programmatore attraverso la modalita' reale del **DOS**; come gia' sappiamo, gli indirizzi logici compresi tra **FFFFh:0010h** e **FFFFh:FFFFh** esistono solo in presenza di un **PC** dotato di **CPU 80286** o superiore, e di oltre **1** Mb di **RAM**.
Nella terminologia del **DOS**, tutta la memoria rappresentata dal primo Mb di **RAM** viene chiamata **Base Memory** (memoria base); si tratta come sappiamo della sola memoria direttamente indirizzabile in modalita' reale **8086**. La memoria base viene suddivisa in due parti chiamate **Conventional Memory** (memoria convenzionale) e **Upper Memory** (memoria superiore); il confine tra queste due aree di memoria viene delimitato dall'indirizzo fisico **000A0000h** che puo' essere associato all'indirizzo logico **A000h:0000h**. Come gia' sappiamo, il blocco di memoria da **65520** byte immediatamente successivo al primo Mb viene chiamato **High Memory** (memoria alta); questo blocco e' indirizzabile in modalita' reale solo con una **CPU 80286** o superiore (con la linea **A20** abilitata). Tutta la memoria che sta al di sopra della **High Memory** viene chiamata **Extended Memory** (memoria estesa); questa memoria e' direttamente indirizzabile solo in modalita' protetta dalle **CPU 80286** e superiori.

Analizziamo ora le varie aree che nel loro insieme formano la mappa della **RAM** relativa all'ambiente operativo supportato dal **DOS**.

9.3.1 Vettori di interruzione - da 00000000h a 000003FFh

Come e' stato detto in un precedente capitolo, i vettori di interruzione sono degli indirizzi che individuano la posizione in memoria di una serie di procedure (spesso scritte in **Assembly**); queste procedure, rendono disponibili ai programmi in esecuzione, una numerosa serie di servizi di basso livello (livello macchina) offerti dall'hardware del computer.
Sfruttando le nuove conoscenze che abbiamo acquisito in questo capitolo, possiamo aggiungere che ciascun vettore di interruzione e' un indirizzo logico del tipo **Seg:Offset** da **16+16** bit; risulta evidente quindi che questi vettori di interruzione, sono disponibili solo quando le **CPU** della famiglia **80x86**, lavorano in modalita' reale. Proprio per questo motivo, ciascun vettore di interruzione "punta" ad una procedura che deve necessariamente trovarsi nel primo Mb della **RAM**; in questo modo, la procedura stessa puo' essere chiamabile da un programma che opera in modalita' reale **8086**, e che puo' vedere quindi solo **1** Mb di **RAM**.
Generalmente, i vettori di interruzione vengono installati in fase di avvio del computer; una serie piuttosto numerosa di vettori di interruzione, viene installata principalmente dal **DOS** e dal **BIOS** del **PC**. L'installazione di un vettore di interruzione consiste innanzi tutto nel caricamento in memoria della relativa procedura, chiamata **ISR** o **Interrupt Service Routine** (procedura di servizio dell'interruzione); l'indirizzo di questa procedura, viene poi sistemato nell'apposita area della **RAM**, riservata ai vettori di interruzione.
In seguito ad una serie di convenzioni stabilite dai produttori di hardware e di **SO**, e' stato deciso che i **PC** debbano riservare un'area della **RAM** sufficiente a contenere **256** vettori di interruzione; ciascun vettore e' un indirizzo logico **Seg:Offset** che richiede **16+16=32** bit (**4** byte), e quindi, l'area complessiva della **RAM** riservata ai vettori di interruzione e' pari a:
```
256 * 4 = 1024 byte = 400h byte
```
Quest'area deve essere rigorosamente posizionata nella parte iniziale della **RAM** compresa tra gli indirizzi fisici **00000000h** e **000003FFh**; ciascun vettore di interruzione viene identificato da un indice compreso tra **0** e **255** (in esadecimale, tra **00h** e **FFh**). Come si puo' facilmente intuire, molti

di questi indici sono riservati rigorosamente al **DOS** e al **BIOS** del **PC**; il vettore n. **10h**, ad esempio, e' riservato ai servizi offerti dal **BIOS** della scheda video. Se vogliamo usufruire di questi servizi, non dobbiamo fare altro che chiamare la procedura che si trova all'indirizzo associato al vettore di interruzione n. **10h**; la fase di chiamata di un vettore di interruzione, effettuata da un programma in esecuzione, prende il nome di **interruzione software**. I servizi offerti dai vettori di interruzione, vengono richiesti anche dalle periferiche che hanno la necessita' di comunicare con la **CPU**; in questo caso si parla di **interruzione hardware** (questo argomento e' stato gia' trattato Capitolo 7).
In relazione al **DOS**, i vettori di interruzione assumono una importanza fondamentale; infatti, i programmi in esecuzione comunicano con il **DOS** proprio attraverso questi vettori. Se un programma vuole ad esempio creare un file sul disco rigido, non deve fare altro che chiamare un apposito vettore di interruzione del **DOS**; il vettore di interruzione principale per i servizi **DOS** e' sicuramente il n. **21h**, che proprio per questo motivo, viene chiamato **vettore dei servizi DOS**.

9.3.2 Area dati ROM-BIOS - da 00000400h a 000004FFh

Nel Capitolo 7 e' stato detto che quando si accende il computer, il controllo passa ad una serie di programmi memorizzati nella **ROM** del **PC**; questi programmi comprendono in particolare il **POST** che esegue l'autodiagnosi relativa a tutto l'hardware del computer.
Uno dei compiti fondamentali svolti dai programmi della **ROM** consiste nella raccolta di una serie di importanti informazioni relative alle caratteristiche generali dell'hardware del **PC**; queste informazioni vengono messe a disposizione dei programmi e comprendono ad esempio gli indirizzi delle porte seriali e parallele, lo stato corrente dell'hard disk e del floppy disk, la modalita' video corrente, etc.
Tutte queste informazioni vengono inserite in un'area della **RAM** formata da **256** byte (**100h** byte); per convenzione quest'area deve essere rigorosamente compresa tra gli indirizzi fisici **00000400h** e **000004FFh**.

9.3.3 Area comunicazioni del DOS - da 00000500h a 000006FFh

Come gia' sappiamo, l'ultimo compito svolto dai programmi della **ROM** in fase di avvio del computer, consiste nel cercare ed eventualmente eseguire un programma chiamato **boot loader**; il **boot loader** ha l'importante compito di caricare in memoria il **SO** che nel nostro caso e' il **DOS**.
Una volta che il **DOS** ha ricevuto il controllo, esegue a sua volta una serie di inizializzazioni; in questa fase il **DOS** raccoglie anche una serie di importanti informazioni globali relative al **SO**.
Tutte queste informazioni vengono messe a disposizione dei programmi in un'area della **RAM** formata da **512** byte (**200h** byte); per convenzione quest'area deve essere rigorosamente compresa tra gli indirizzi fisici **00000500h** e **000006FFh**.

9.3.4 Kernel del DOS, Device Drivers, etc - da 00000700h a ????????h

In quest'area viene caricato in particolare il kernel del **DOS** formato dai due file **IO.SYS** e **MSDOS.SYS**; sempre in quest'area trovano posto svariati **device drivers** usati dal **DOS** e alcuni buffers interni (aree usate dal **DOS** per depositare informazioni temporanee).
Come si puo' notare in Figura 7, quest'area parte rigorosamente dall'indirizzo fisico **00000700h**, e assume una dimensione variabile; cio' e' dovuto al fatto che a seconda della configurazione del proprio computer, e' possibile ridurre le dimensioni di quest'area spostando porzioni del **DOS** in memoria alta o nella memoria superiore e liberando cosi' spazio nella memoria convenzionale. Sui computers dotati di un vero **DOS** (quindi non un emulatore), e' presente in **C:** un file chiamato **CONFIG.SYS**; all'interno di questo file si possono trovare i comandi che permettono appunto di liberare spazio nella memoria convenzionale. Il comando:

```
DOS=UMB
```
se e' presente permette di spostare parti del **DOS** in memoria superiore; la sigla **UMB** sta per **Upper Memory Blocks** (blocchi di memoria superiore).
Il comando:
```
DOS=HIGH
```
se e' presente permette di spostare parti del **DOS** in memoria alta.
In generale, l'area della **RAM** destinata ad ospitare il kernel del **DOS**, i device drivers e i buffers interni occupa alcune decine di Kb.

9.3.5 Disponibile per i programmi DOS - da ????????h a 000A0000h

Tutta l'area rimanente nella memoria convenzionale e' disponibile per i programmi **DOS**; quest'area inizia quindi dalla fine del blocco precedentemente descritto, e termina all'indirizzo fisico **0009FFFFh**. L'indirizzo fisico successivo e' **000A0000h** che tradotto in base **10** corrisponde a **655360** (**640*1024**) e rappresenta la tristemente famosa barriera dei **640** Kb; un qualunque programma **DOS** per poter essere caricato in memoria ed eseguito, deve avere dimensioni non superiori a **640** Kb. Commentando questo aspetto, l'allora semi sconosciuto **Bill Gates** pronuncio' una famosa frase dicendo che: **"640 Kb rappresentano una quantita' enorme di memoria, piu' che sufficiente per far girare qualunque applicazione presente e futura"!**
In realta' un programma da **640** Kb e' troppo grande in quanto abbiamo visto che i primi Kb della **RAM** sono riservati ai vettori di interruzione, all'area dati della **ROM BIOS**, etc; tolte queste aree riservate restano a disposizione per i programmi circa **600** Kb effettivi!

9.3.6 Buffer video - modo grafico - da 000A0000h a 000AFFFFh

L'indirizzo fisico **000A0000h** segna l'inizio della memoria superiore; teoricamente la memoria superiore e' un'area riservata, non utilizzabile quindi in modo diretto dai programmi **DOS**.
In questo capitolo abbiamo visto che una **CPU** che opera in modalita' reale **8086** e' in grado di indirizzare in modo diretto solo **1** Mb di **RAM** (memoria base); questo significa che qualsiasi altra memoria esterna, per poter essere accessibile deve essere mappata in qualche zona della memoria base (**I/O Memory Mapped**). Lo scopo principale della memoria superiore e' proprio quello di contenere svariati buffers nei quali vengono mappate le memorie esterne come la memoria video, la **ROM BIOS**, etc; questi buffers sono aree di scambio attraverso le quali un programma **DOS** puo' comunicare con le memorie esterne.
L'area compresa tra gli indirizzi fisici **000A0000h** e **000AFFFFh** viene utilizzata per mappare la memoria video in modalita' grafica; quest'area ha una dimensione pari a:
```
000AFFFFh - 000A0000h + 1h = 10000h byte = 65536 byte = 64 Kb
```
Attraverso quest'area un programma **DOS** puo' quindi leggere o scrivere in un blocco da **64** Kb della memoria video grafica; tutto cio' ci fa' intuire che, com'era prevedibile, la segmentazione a **64** Kb della modalita' reale si ripercuote anche sulle memorie esterne.
Un buffer per una memoria esterna puo' essere paragonato ad un fotogramma di una pellicola cinematografica; il proiettore fa' scorrere la pellicola mostrando agli spettatori la sequenza dei vari fotogrammi. Allo stesso modo il programmatore puo' richiedere la mappatura nel buffer video della porzione desiderata della memoria video; questa tecnica consente di accedere dal buffer video a tutta la memoria video disponibile. Proprio per questo motivo, un buffer come quello descritto prende anche il nome di **frame window** (finestra fotogramma) o **frame buffer** (buffer fotogramma).

9.3.7 Buffer video - modo testo in b / n - da **000B0000h** a **000B7FFFh**

L'area della **RAM** compresa tra gli indirizzi fisici **000B0000h** e **000B7FFFh** contiene il buffer per l'**I/O** con la memoria video in modalita' testo per i vecchi monitor in bianco e nero; si tratta quindi di un buffer da **8000h** byte, cioe' **32768** byte (**32** Kb).

9.3.8 Buffer video - modo testo a colori - da **000B8000h** a **000BFFFFh**

L'area della **RAM** compresa tra gli indirizzi fisici **000B8000h** e **000BFFFFh** contiene il buffer per l'**I/O** con la memoria video in modalita' testo a colori; si tratta anche in questo caso di un buffer da **8000h** byte, cioe' **32768** byte (**32** Kb).

9.3.9 Video ROM BIOS - da **000C0000h** a **000CFFFFh**

In quest'area da **64** Kb viene mappata la **ROM BIOS** delle schede video che contiene una numerosa serie di procedure per l'accesso a basso livello all'hardware della scheda video; le schede video sono ben presto diventate talmente potenti e sofisticate da richiedere un loro specifico **BIOS** che consente ai **SO** di gestire al meglio queste periferiche.

9.3.10 Disponibile per altri BIOS e buffers - da **000D0000h** a **000EFFFFh**

Quest'area da **128** Kb e' disponibile per ospitare ulteriori buffers per il collegamento con altre memorie esterne; in particolare, in quest'area viene creata la **frame window** per l'accesso alle espansioni di memoria che si utilizzavano ai tempi delle **CPU 8086**.
Puo' capitare frequentemente che in quest'area rimangano dei blocchi liberi di memoria; questi blocchi liberi possono essere utilizzati dai programmi **DOS**, e in certi casi e' persino possibile far girare programmi **DOS** in memoria superiore. A tale proposito e' necessario disporre di una **CPU 80386** o superiore e di un cosiddetto **Memory Manager** (gestore della memoria) che permetta di indirizzare in modalita' reale queste aree riservate; nel mondo del **DOS** il memory manager piu' famoso e' senz'altro **EMM386.EXE** che viene fornito insieme allo stesso **SO**.

9.3.11 PC ROM BIOS - da **000F0000h** a **000FFFFFh**
In quest'area da **64** Kb viene mappata la **ROM BIOS** principale del **PC**; in questo modo i programmi **DOS** possono usufruire di numerosi servizi offerti dal **BIOS** per accedere a basso livello all'hardware del computer.
Come si puo' notare, quest'area comprende anche i famosi **4095** segmenti di memoria incompleti; riservando questi segmenti al **BIOS**, si evita che i normali programmi **DOS** possano incappare nel **wrap around**.

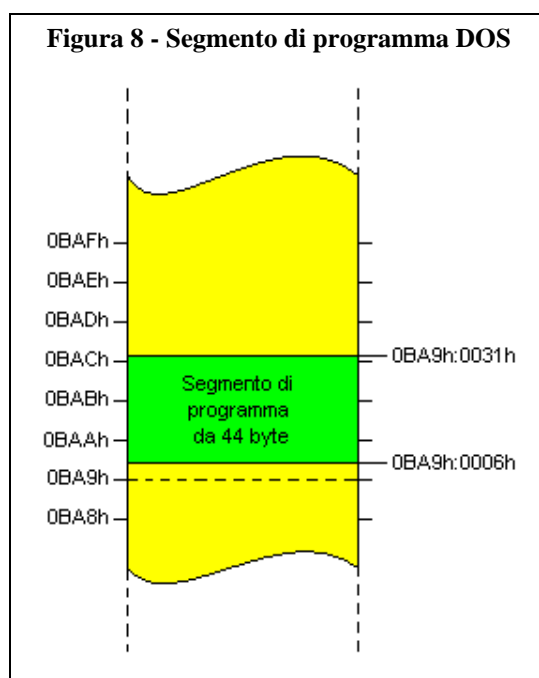
9.4 Organizzazione della memoria sotto DOS

Per motivi di efficienza il **DOS** suddivide idealmente la **RAM** in paragrafi; in questo modo, la memoria base del computer (cioe' il primo Mb) risulta suddivisa come gia' sappiamo in **65536** blocchi da **16** byte ciascuno. Il paragrafo rappresenta anche la **granularita'** della memoria **DOS**, cioe' la quantita' minima di memoria che il **DOS** puo' gestire; cio' significa che un qualunque blocco di memoria **DOS** ha una dimensione che e' sempre un multiplo intero di **16** byte (**16**, **32**, **48**, **64**, etc).
Tutti i blocchi di memoria **DOS** sono sempre allineati al paragrafo; di conseguenza, l'indirizzo logico iniziale di un blocco di memoria **DOS** e' sempre del tipo **XXXXh:0000h**. Cio' permette al **DOS** di riferirsi a questi blocchi attraverso la sola componente **Seg** dell'indirizzo iniziale; la componente **Offset** iniziale infatti e' ovviamente **0000h**.

Supponiamo ad esempio di richiedere al **DOS** un blocco da **420** paragrafi di memoria (**6720** byte); se il **DOS** accetta la richiesta, ci restituisce un valore a **16** bit che rappresenta appunto la componente **Seg** dell'indirizzo iniziale del blocco di memoria richiesto. Supponiamo a tale proposito che il **DOS** ci restituisca il valore **0BF2h**; tenendo presente che **6720** in esadecimale si scrive **1A40h**, possiamo dire che il blocco di memoria messoci a disposizione dal **DOS** e' compreso tra gli indirizzi logici **0BF2h:0000h** e **0BF2h:1A3Fh** per un totale appunto di **1A40h** byte (**6720** byte).
Da queste considerazioni risulta evidente che la dimensione minima di un blocco di memoria **DOS** e' pari a **1** paragrafo (**16** byte), mentre la dimensione massima e' pari a **4096** paragrafi (**65536** byte = **64** Kb); questo limite massimo e' ovviamente una diretta conseguenza della modalita' reale **8086** che non permette di esprimere un offset maggiore di **FFFFh**.
Se abbiamo bisogno di un blocco di memoria piu' grande di **64** Kb, siamo costretti a richiedere al **DOS** due o piu' blocchi di memoria, ciascuno dei quali non puo' superare i **64** Kb; in certi casi questa situazione puo' creare parecchi fastidi al programmatore che si trova costretto a dover saltare continuamente da un blocco di memoria ad un altro.

9.5 Indirizzamento di un programma DOS

Un programma **DOS** viene suddiviso in blocchi chiamati **segmenti di programma** (da non confondere con i **segmenti di memoria**); nel caso piu' semplice e intuitivo, un programma **DOS** e' formato da due segmenti chiamati **Data Segment** (segmento dati) e **Code Segment** (segmento di codice). Come al solito, il segmento dati contiene i dati temporanei e permanenti del programma; il segmento di codice contiene invece le istruzioni che devono elaborare i dati stessi.
Al programmatore viene data la possibilita' di stabilire il tipo di allineamento in memoria per ogni segmento di programma; come vedremo nei capitoli successivi, esiste la possibilita' di richiedere un allineamento ad indirizzi multipli interi di **2**, di **4**, di **16**, etc, oppure nessun allineamento (cioe' un allineamento ad un indirizzo qualunque). E' chiaro pero' che un qualunque segmento di programma, nel rispetto della modalita' reale **8086**, deve essere necessariamente indirizzato attraverso coppie logiche del tipo **Seg:Offset**; di conseguenza, la componente **Offset** relativa ad una informazione presente in un segmento di programma, verra' sempre calcolata rispetto ad una componente **Seg** che rappresenta, come sappiamo, un segmento di memoria allineato al paragrafo. La Figura 8 chiarisce questo importantissimo concetto, che verra' comunque approfondito in un altro capitolo.

Figura 8 - Segmento di programma DOS

Questa figura si riferisce ad un generico segmento di programma da **44** byte (**2Ch** byte), per il quale abbiamo richiesto un allineamento qualunque in memoria; in questo esempio supponiamo che dopo il caricamento in memoria, questo segmento di programma venga posizionato in modo da partire dall'indirizzo fisico a **20** bit **0BA96h**. Questo indirizzo fisico corrisponde all'indirizzo logico normalizzato **0BA9h:0006h**, la cui componente **Seg** e' associata, come sappiamo, all'indirizzo fisico multiplo di **16** (**0BA90h**) che piu' si avvicina per difetto a **0BA96h**; possiamo dire allora che la **CPU** indirizza questo segmento di programma, utilizzando coppie logiche **Seg:Offset** con la componente **Seg** che vale **0BA9h**.
La componente **Offset** di conseguenza partira' dal valore minimo **0006h**; attraverso la componente **Offset** possiamo muoverci all'interno del segmento di programma accedendo a tutti i suoi **44** byte.
Il primo byte del segmento di programma si trova all'offset **6**, mentre l'ultimo byte, cioe' il numero **44** si viene a trovare all'offset:
`6 + (44 - 1) = 6 + 43 = 49 = 31h`
Riassumendo, possiamo dire che il primo byte del segmento di programma si trova all'indirizzo logico **0BA9h:0006h**, mentre l'ultimo byte si trova all'indirizzo logico **0BA9h:0031h**.
Come si puo' notare in Figura 8, il nostro segmento di programma termina poco oltre l'inizio del segmento di memoria n. **0BACh**; osservando infatti che l'offset **31h** e' formato da **3** paragrafi (**30h**) piu' **1** byte, ricaviamo subito:
`0BA9h:0031h = (0BA9h + 3h):0001h = 0BACh:0001h`
Questi concetti possono dare l'impressione di essere un po' ostici da capire; in realta', con un po' di esercizio ci si rende conto che si tratta di aspetti abbastanza elementari.

Tutte le considerazioni precedentemente esposte, hanno una importante implicazione che, come al solito, e' una diretta conseguenza della modalita' reale **8086**; osserviamo innanzi tutto che una componente **Offset** di un indirizzo logico puo' assumere tutti i valori compresi tra **0000h** e **FFFFh**.
Il segmento di programma di Figura 8 inizia dall'offset **0006h** e termina all'offset **0031h**; se volessimo aggiungere informazioni a questo segmento, le sue dimensioni potrebbero crescere liberamente sino ad arrivare all'offset **FFFFh**. La dimensione massima di questo segmento di programma sarebbe quindi:
`FFFFh - 0006h + 1h = FFFAh byte = 65530 byte`
Nella migliore delle ipotesi, possiamo allineare un segmento di programma al paragrafo, in modo che la sua componente **Offset** parta sicuramente da **0000h**; in questo caso, l'offset puo' variare da **0000h** a **FFFFh**, e il segmento di programma raggiungerebbe la dimensione massima possibile, pari a:
`FFFFh - 0000h + 1h = 10000h byte = 65536 byte = 64 Kb`
Tutto cio' significa che un qualunque segmento di programma, non puo' superare la dimensione massima di **64** Kb!
Se il nostro programma ha bisogno di meno di **64** Kb di dati e meno di **64** Kb di codice, allora possiamo ritenerci fortunati; in questo caso infatti, dobbiamo comunicare alla **CPU** la componente **Seg** relativa all'unico blocco codice e la componente **Seg** relativa all'unico blocco dati. Supponiamo ad esempio di chiamare queste due componenti **SegCodice** e **SegDati**; esistendo un solo blocco codice e un solo blocco dati, queste due componenti restano fisse per tutta la fase di esecuzione del programma. Se vogliamo allora accedere a un dato, abbiamo la possibilita' di comunicare alla **CPU** la sola componente **Offset** dell'indirizzo del dato stesso; la **CPU** associa automaticamente **Offset** a **SegDati** ottenendo cosi' l'indirizzo logico **SegDati:Offset**.
Un indirizzo formato dalla sola componente **Offset** a **16** bit viene chiamato **indirizzo NEAR** (indirizzo vicino); gli indirizzi **NEAR** ci permettono di risparmiare memoria, e vengono gestiti molto piu' velocemente dalla **CPU**.

Se pero' il nostro programma ha bisogno di piu' di **64** Kb di dati e/o piu' di **64** Kb di codice, allora entriamo nell'inferno della segmentazione a **64** Kb; in questo caso infatti, il nostro programma deve essere suddiviso in due o piu' segmenti di codice e/o due o piu' segmenti di dati. Supponiamo ad

esempio che il nostro programma abbia due segmenti di dati associati alle due componenti **SegDati1** e **SegDati2**; in questo caso, se vogliamo accedere a un dato, dobbiamo comunicare alla **CPU**, non solo la componente **Offset**, ma anche la componente **SegDati1** o **SegDati2** che identifica il segmento di appartenenza del dato stesso.
Un indirizzo formato da una coppia completa **Seg:Offset** a **16+16** bit viene chiamato **indirizzo FAR** (indirizzo lontano); gli indirizzi **FAR** occupano il doppio della memoria necessaria per un indirizzo **NEAR**, e quindi risultano piu' lenti da gestire per la **CPU**.
I linguaggi di programmazione di alto livello rivolti ai programmatori esperti, mettono a disposizione le cosiddette **variabili puntatore** che sono destinate a contenere proprio indirizzi di tipo **NEAR** o **FAR**; nel linguaggio **C** ad esempio, la definizione:

```
char near *ptcn;
```

crea un dato intero senza segno a **16** bit chiamato **ptcn** e destinato a contenere l'indirizzo **NEAR** di un dato di tipo **char** (intero con segno a **8** bit). La variabile **ptcn** viene chiamata **puntatore NEAR**.
Analogamente, la definizione:

```
char far *ptcf;
```

crea un dato intero senza segno a **32** bit chiamato **ptcf** e destinato a contenere l'indirizzo **FAR** di un dato di tipo **char** (intero con segno a **8** bit). La variabile **ptcf** viene chiamata **puntatore FAR**.

9.6 Supporto del DOS sui sistemi operativi per la modalita' protetta

Esistono numerosi **SO** destinati a supportare la modalita' protetta delle **CPU 80286** e superiori; tra i piu' famosi si possono citare **Windows**, **OS/2**, **UNIX** e **Linux**. Le versioni attuali di questi **SO**, richiedono almeno una **CPU 80386**, in modo da poter usufruire di un supporto piu' avanzato della modalita' protetta; tutti questi **SO**, sono di tipo **multitasking**, e quindi sono in grado di eseguire piu' programmi contemporaneamente. Naturalmente, con un'unica **CPU** a disposizione, e' possibile eseguire un solo programma alla volta; per ottenere allora il **multitasking**, si utilizza un noto espediente chiamato **time sharing** (ripartizione del tempo). In sostanza, il **SO** fa' girare uno alla volta tutti i programmi, assegnando a ciascuno di essi un piccolissimo intervallo di tempo di esecuzione (una frazione di secondo); in questo modo, l'utente ha l'impressione che i vari programmi stiano girando contemporaneamente.

I **SO** della famiglia **Windows** offrono il pieno supporto dei programmi scritti per la modalita' reale del **DOS**; questo supporto varia da versione a versione di **Windows**. Nel caso di **Windows 95** e **Windows 98** si ha a disposizione un vero **DOS**; come e' stato ampiamente dimostrato da alcuni hackers, in realta' queste due versioni di **Windows** sono delle sofisticate interfacce grafiche che si appoggiano sul **DOS**!
All'interno di **Windows 95** e **Windows 98** si puo' attivare il tipico ambiente **DOS** selezionando il menu **Start - Programmi - Prompt di MS-DOS**; in questo modo viene aperta una finestra (**Dos Box**) che simula la caratteristica schermata testuale del **DOS** con tanto di cursore lampeggiante.
Nella cartella **C:** si puo' constatare la presenza dei file nascosti **IO.SYS**, **MSDOS.SYS** e **COMMAND.COM**; la cartella riservata ai vari file del **DOS** e' **C:\WINDOWS\COMMAND**. La presenza di un vero **DOS** permette anche di richiedere il riavvio del computer in modalita' **MS-DOS**; in questo caso ci ritroviamo in un ambiente **DOS** vero e proprio.

Nel caso invece di **Windows XP**, viene fornito un emulatore **DOS**; questo emulatore fa' egregiamente il suo dovere ma risulta essere particolarmente lento; con questa versione di **Windows** ovviamente non e' possibile riavviare il computer in modalita' **DOS**.

Il **DOS** e' un **SO monotasking**, ed e' in grado quindi di eseguire un solo programma per volta; l'unico programma in esecuzione ha tutto il computer a sua completa disposizione e puo' accedere liberamente e direttamente a tutto l'hardware disponibile. Ci si puo' chiedere allora come faccia un

SO come **Windows** ad eseguire addirittura due o piu' programmi **DOS** contemporaneamente ad altri programmi **Windows** che tra l'altro girano anche in modalita' protetta.
Per poter fare una cosa del genere questi **SO** sfruttano una caratteristica veramente rivoluzionaria delle **CPU 80386** e superiori, rappresentata dalla cosiddetta **modalita' virtuale 8086** (**V86**); si tratta di una potente caratteristica che permette alla **CPU** di eseguire programmi **DOS** senza uscire dalla modalita' protetta!
Nel caso dell'**80286**, se ci troviamo in modalita' protetta e vogliamo eseguire un programma **DOS**, dobbiamo prima tornare in modalita' reale; queste continue commutazioni tra modalita' protetta e modalita' reale provocano tra l'altro un sensibile rallentamento nel funzionamento del computer. La modalita' **V86** delle **CPU 80386** o superiori crea una **macchina virtuale 8086** costituita da un'area di memoria virtuale da **1** Mb; un programma **DOS** gira in questo Mb virtuale credendo di trovarsi in un tipico ambiente **DOS**. Il Mb virtuale creato dalla modalita' **V86** puo' trovarsi in qualunque area della memoria; l'aspetto straordinario della modalita' **V86** e' che grazie al **time sharing**, un **SO multitasking** puo' creare piu' macchine virtuali facendo girare piu' programmi **DOS** contemporaneamente!
Come si puo' facilmente intuire, in un **SO** di tipo **multitasking** tutti i tentativi di accesso all'hardware da parte di un programma in esecuzione vengono filtrati dal **SO** stesso; in questo modo e' possibile fare in modo che piu' programmi in esecuzione possano accedere alla stessa periferica senza interferenze reciproche.

Capitolo 10 - Architettura interna della CPU

Il compito fondamentale della **CPU** e' quello di eseguire un programma; di conseguenza, la struttura interna della **CPU** e' strettamente legata alla struttura dei programmi da eseguire. La prima cosa che dobbiamo fare consiste quindi nell'analizzare in dettaglio la struttura che caratterizza un programma destinato alla piattaforma hardware **80x86**; in particolare, nella sezione **Assembly Base** ci occuperemo della struttura di un programma destinato ad essere eseguito in modalita' reale **8086**.

10.1 Struttura generale di un programma

Nei precedenti capitoli e' stato ampiamente detto che un programma e' formato da un insieme di dati e istruzioni; per motivi di stile e soprattutto di efficienza, e' consigliabile quindi suddividere un programma in due blocchi fondamentali chiamati **blocco codice** e **blocco dati**. Come gia' sappiamo, il blocco dati contiene i vari dati temporanei e permanenti di un programma; il blocco codice contiene invece le istruzioni destinate ad elaborare i dati stessi.
L'aspetto interessante da analizzare riguarda proprio la distinzione che viene fatta tra dati temporanei e dati permanenti; si tratta di una distinzione talmente importante da suggerire, come vedremo tra breve, una ulteriore suddivisione del blocco dati in due parti destinate a contenere appunto, una i dati temporanei, e l'altra i dati permanenti.
I dati permanenti sono cosi' chiamati in quanto permangono (cioe' esistono) in memoria per tutta la fase di esecuzione di un programma; tra i dati permanenti piu' importanti si possono citare ad esempio i dati in input e i dati in output del programma.
I dati temporanei sono cosi' chiamati in quanto la loro esistenza in memoria e' necessaria solo per un periodo limitato di tempo; una situazione del genere si verifica ad esempio quando abbiamo bisogno temporaneamente di una certa quantita' di memoria per salvare un risultato parziale di una operazione matematica.
In generale, a ciascun dato temporaneo o permanente, viene assegnata una apposita locazione di memoria; la fase di assegnamento della memoria prende il nome di **allocazione**. La distinzione tra dati temporanei e dati permanenti scaturisce proprio dalla diversa gestione della memoria destinata a questi due tipi di dati; infatti, come si puo' facilmente intuire, la memoria destinata ad un dato permanente deve esistere per tutta la fase di esecuzione di un programma, mentre la memoria destinata ad un dato temporaneo e' necessaria solo per un intervallo di tempo limitato.

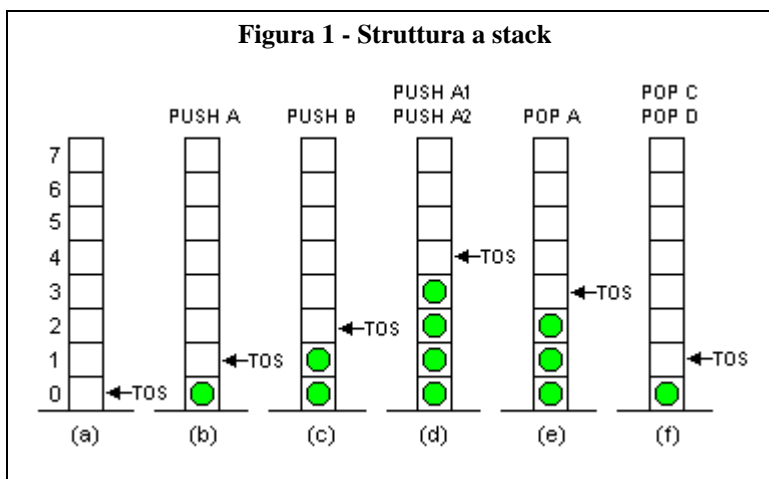
10.1.1 Il Data Segment di un programma

Dalle considerazioni appena esposte, si deduce subito che per i dati permanenti di un programma conviene predisporre un apposito blocco di memoria che esiste per tutta la fase di esecuzione, e le cui dimensioni sono sufficienti per contenere l'insieme dei dati stessi; questo blocco viene internamente suddiviso in locazioni ciascuna delle quali e' destinata a contenere uno dei dati del programma.
L'indirizzo di ciascuna locazione rimane statico, e cioe' fisso per tutta la fase di esecuzione, e per questo motivo si dice anche che la memoria per un dato permanente viene **allocata staticamente**; sempre per lo stesso motivo, i dati permanenti vengono anche chiamati **dati statici**.
Conoscendo l'indirizzo di un dato statico, possiamo accedere alla corrispondente locazione di memoria modificandone il contenuto; un dato statico il cui contenuto puo' essere modificato (variato), viene anche definito **variabile statica**.
Il blocco di memoria destinato a contenere i dati statici forma un segmento di programma che prende il nome di **Data Segment** (segmento dati).

10.1.2 Lo Stack Segment di un programma

Teoricamente per i dati temporanei di un programma si potrebbe procedere come per i dati statici; in questo modo pero' si andrebbe incontro ad un gigantesco spreco di memoria. Osserviamo infatti che e' praticamente impossibile che i dati temporanei di un programma siano necessari tutti nello stesso istante di tempo; cio' significa che le locazioni di memoria allocate staticamente per i dati temporanei resterebbero inutilizzate per buona parte della fase di esecuzione.
Nel corso della sua "esistenza" un programma puo' arrivare a gestire una quantita' di dati temporanei, tale da richiedere decine di migliaia di byte di memoria; istante per istante pero' questo stesso programma puo' avere bisogno di una quantita' di dati temporanei che occupano mediamente appena qualche centinaio di byte di memoria!
Questa situazione richiede un attento studio volto ad individuare il metodo piu' semplice ed efficiente per la gestione dei dati temporanei; questo metodo ci deve permettere in sostanza di ridurre al minimo indispensabile le dimensioni del blocco di memoria destinato a questo particolare tipo di dati.
Per individuare la strada da seguire, possiamo fare riferimento alle considerazioni esposte in precedenza, in base alle quali una locazione di memoria destinata ad un dato temporaneo, viene sfruttata solo per un periodo limitato di tempo; possiamo pensare allora di "creare" questa locazione quando ne abbiamo bisogno, "distruggendola" poi quando non ci serve piu'. Questo tipo di gestione "al volo" (cioe' in fase di esecuzione) della memoria viene definito **dinamico**; la fase di creazione al volo di una locazione di memoria viene definita **allocazione dinamica**, mentre la fase di distruzione al volo della stessa locazione viene definita **deallocazione dinamica**.
L'unico svantaggio di questo sistema e' legato alla relativa lentezza delle numerosissime operazioni di allocazione e deallocazione; questo unico svantaggio scompare pero' davanti al fatto che con questa tecnica possiamo arrivare ad ottenere un risparmio di memoria veramente colossale (poche centinaia di byte anziche' decine di migliaia di byte).

Esistono diverse strutture dati che permettono di implementare un sistema di gestione dei dati temporanei; la piu' semplice e la piu' efficiente tra queste strutture e' pero' la cosiddetta **pila** (in inglese **stack**). Come si intuisce dal nome, la pila e' un contenitore all'interno del quale possiamo disporre (impilare) uno sull'altro una serie di oggetti, tutti della stessa natura; possiamo avere cosi' una pila di sedie, una pila di scatole, una pila di numeri interi a **16** bit e cosi' via.
La gestione delle strutture a pila assume una importanza enorme nel mondo del computer, per cui questo argomento deve essere necessariamente analizzato in modo approfondito; a tale proposito ci serviamo della Figura 1 che illustra il funzionamento di una pila di palline.

Figura 1 - Struttura a stack

Osserviamo subito che il contenitore della pila e' dotato di una serie di ripiani o locazioni; ciascuna locazione viene univocamente individuata da un apposito indice. In Figura 1 abbiamo quindi una pila da **8** locazioni numerate da **0** a **7**; ciascuna locazione puo' contenere una e una sola pallina.
Appare anche evidente che la pila, per sua stessa natura e' dotata di una sola imboccatura attraverso la quale possiamo inserire o estrarre le palline; e' chiaro allora che le varie palline inserite, andranno a disporsi non in modo casuale, ma in modo ordinato, e cioe' una sopra l'altra. Analogamente, durante la fase di estrazione, le palline escono non in modo casuale, ma in modo ordinato, e cioe' a partire da quella che si trova in cima; la pallina che si trova in cima alla pila e' ovviamente quella che era stata inserita per ultima.
In generale, in una struttura a pila il primo oggetto che viene estratto coincide con l'ultimo oggetto che era stato inserito; per questo motivo si dice che la pila e' una struttura di tipo **LIFO** o **Last In, First Out** (l'ultimo entrato e' anche il primo che verra' estratto).

Le due operazioni fondamentali che possiamo compiere con una pila sono l'**inserimento** di un oggetto e l'**estrazione** di un oggetto; l'operazione di inserimento viene chiamata **PUSH**, mentre l'operazione di estrazione viene chiamata **POP**.
L'istruzione (o **operatore**) **PUSH** ha bisogno di una locazione (o **operando**) chiamata **sorgente**; l'operando sorgente dice a **PUSH** dove prelevare la pallina da inserire in cima alla pila.
L'istruzione (o **operatore**) **POP** ha bisogno di una locazione (o **operando**) chiamata **destinazione**; l'operando destinazione dice a **POP** dove mettere la pallina appena estratta dalla pila.
Per gestire correttamente tutta questa situazione abbiamo bisogno di una serie di importanti informazioni; in particolare dobbiamo conoscere la dimensione della pila e la posizione (indice) della cima della pila stessa. La cima della pila (**Top Of Stack**) viene indicata con la sigla **TOS**; in Figura 1a vediamo che inizialmente la pila e' vuota, e questa situazione e' rappresentata chiaramente dalla condizione **TOS=0**. L'indice **0** rappresenta quindi il fondo della pila; l'indice **7** e' invece quello della locazione che si trova all'imbocco della pila stessa.

Osserviamo ora in Figura 1b quello che succede in seguito all'istruzione:
```
PUSH A
```
Questa istruzione comporta due fasi distinte:

1) La pallina contenuta nella locazione sorgente **A** viene inserita nella pila all'indice **TOS=0**.
2) Il **TOS** viene incrementato di **1** e si ottiene **TOS=1**.

La Figura 1c mostra l'inserimento nella pila di una seconda pallina prelevata dalla sorgente **B**; questa pallina si posiziona all'indice **1** immediatamente sopra la prima pallina, mentre **TOS** si posiziona sull'indice **2**.
La Figura 1d mostra l'inserimento nella pila due ulteriori palline prelevate dalle sorgenti **A1** e **A2**; questa fase richiede naturalmente due istruzioni **PUSH** che portano **TOS** prima a **3** e poi a **4**.

In Figura 1e vediamo quello che succede in seguito all'istruzione:
```
POP A
```
Questa istruzione comporta due fasi distinte:

1) Il **TOS** viene decrementato di **1** e si ottiene **TOS=3**.
2) La pallina contenuta all'indice **TOS=3** viene prelevata e sistemata nella locazione destinazione **A**.

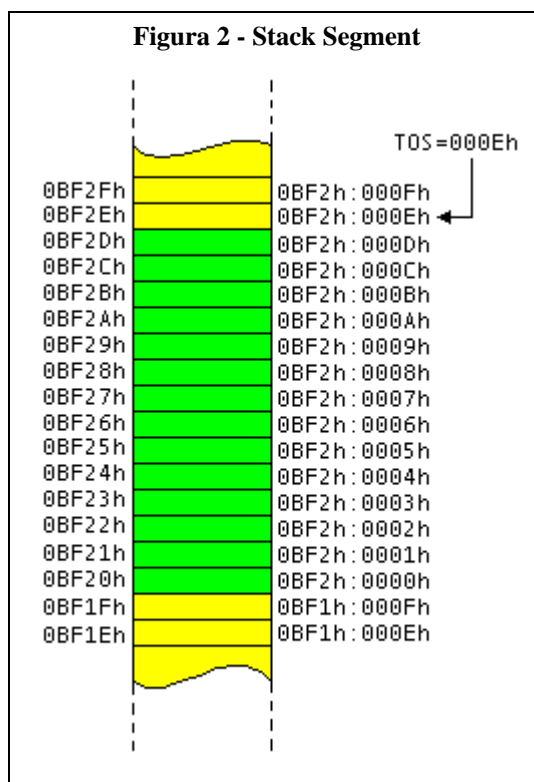
La Figura 1f mostra infine l'estrazione dalla pila di due ulteriori palline da trasferire alle locazioni destinazione **C** e **D**; questa fase richiede naturalmente due istruzioni **POP** che portano **TOS** prima a **2** e poi a **1**.

Se si vuole una gestione piu' sofisticata della pila bisogna prevedere anche il controllo degli errori; osserviamo a tale proposito che l'istruzione **PUSH** puo' essere eseguita solo se **TOS** e' minore di **8**, mentre l'istruzione **POP** puo' essere eseguita solo se **TOS** e' maggiore di **0**.
La Figura 1 ci offre la possibilita' di dedurre una regola fondamentale che si applica alle strutture a pila: `in una struttura a pila, tutte le operazioni di estrazione devono essere effettuate in ordine esattamente inverso alle operazioni di inserimento.`
Nel caso di Figura 1 questa regola puo' sembrare del tutto ovvia; in Figura 1d ad esempio si vede subito che non e' assolutamente possibile estrarre la pallina di indice **2** che si trova sotto la pallina di indice **3**. Prima di estrarre quindi la pallina di indice **2**, dobbiamo estrarre la pallina di indice **3**; nel caso dello stack di un programma vedremo pero' che la situazione non e' cosi' ovvia come sembra.
Possiamo enunciare la precedente regola dicendo anche che: `in una struttura a pila, il numero complessivo delle operazioni di inserimento deve essere perfettamente bilanciato dal numero complessivo delle operazioni di estrazione.`
Se questa regola non viene rispettata ci si ritrova con uno stack che viene definito **sbilanciato**; nel caso di un programma, lo stack sbilanciato porta come vedremo in seguito, ad un sicuro crash.
Un altro aspetto evidente che emerge dalla Figura 1 riguarda il fatto che se una pallina viene prelevata dalla locazione **A** e viene inserita nello stack all'indice **2**, non e' obbligatorio che in fase di estrazione questa stessa pallina venga rimessa per forza nella locazione **A**; nessuno ci impedisce di estrarre questa pallina dallo stack e di metterla in una locazione **B**.

La struttura a pila mostrata in Figura 1 si presta in modo perfetto per la gestione dei dati temporanei di un programma; questa struttura ci permette infatti di ridurre al minimo indispensabile la quantita' di memoria necessaria per questo tipo di dati.
Se ad esempio, un programma ha bisogno istante per istante di una media di **200** byte di spazio in memoria per le variabili temporanee, non dobbiamo fare altro che allocare staticamente un blocco di memoria non inferiore a **200** byte; all'interno di questo blocco vengono continuamente create e distrutte dinamicamente le locazioni per i dati temporanei.
Indubbiamente, il compito piu' delicato per il programmatore consiste nel corretto calcolo della quantita' di memoria da allocare staticamente per il blocco stack; un blocco troppo grande comporta uno spreco di memoria, mentre un blocco troppo piccolo porta ad un cosiddetto **stack overflow** con conseguente crash del programma. E' chiaro che in caso di dubbio conviene sempre approssimare per eccesso le dimensioni del blocco stack.

Vediamo ora come si applicano in pratica i concetti illustrati in Figura 1, per implementare lo stack di un programma. Come e' stato appena detto, la prima cosa da fare consiste nel determinare la dimensione piu' adatta per questo blocco di memoria; questa informazione rappresenta la quantita' di memoria da allocare staticamente per lo stack.
Il blocco stack di un programma viene gestito come al solito attraverso indirizzi logici del tipo **Seg:Offset**; appare evidente il fatto che in questo caso, il **TOS** non e' altro che la componente **Offset** della locazione che si trova in cima allo stack.
Per motivi di efficienza, la gestione dello stack di un programma e' strettamente legata alla architettura della **CPU** che si sta utilizzando; in base ai concetti esposti nei precedenti capitoli, e' fondamentale che vengano garantiti tutti i requisiti di allineamento in memoria sia per l'indirizzo di partenza del blocco stack, sia per il **TOS**.
Una **CPU** con architettura a **8** bit e' predisposta per gestire uno stack di **BYTE**; in questo caso come sappiamo, non esiste nessun problema di allineamento ne' per l'indirizzo iniziale dello stack, ne' per il **TOS**.
Una **CPU** con architettura a **16** bit e' predisposta per gestire uno stack di **WORD**; in questo caso come sappiamo, conviene allineare ad indirizzi pari sia l'inizio del blocco stack, sia il **TOS**.
Una **CPU** con architettura a **32** bit e' predisposta per gestire uno stack di **DWORD**; in questo caso come sappiamo, conviene allineare ad indirizzi multipli interi di **4** sia l'inizio del blocco stack, sia il

TOS.
Per chiarire questi importanti concetti analizziamo in Figura 2 le caratteristiche del blocco stack di un programma; in particolare, questa figura si riferisce ad uno stack di **WORD**, tipico dei programmi che girano in modalita' reale **8086**.

Figura 2 - Stack Segment

Sulla sinistra del vettore della **RAM** notiamo la presenza dei vari indirizzi fisici a **20** bit; sulla destra dello stesso vettore notiamo la presenza dei corrispondenti indirizzi logici normalizzati.
Osserviamo subito che il blocco stack di Figura 2 (colorato in verde) ha una capienza di **7 WORD** (**14 BYTE**) e inizia in memoria dall'indirizzo fisico **0BF20h** che e' un indirizzo pari; assegnando anche al **TOS** un valore iniziale pari, possiamo facilmente constatare che tutte le **WORD** presenti nello stack si vengono a trovare a loro volta allineate ad indirizzi pari, e possono essere quindi lette o scritte con un unico accesso in memoria da parte delle **CPU** con architettura a **16** bit o superiore.
La Figura 2 evidenzia anche un aspetto importante che riguarda proprio il valore iniziale assegnato al **TOS**; siccome stiamo creando uno stack da **7 WORD**, cioe' da **14 BYTE**, per indicare il fatto che inizialmente lo stack e' vuoto poniamo **TOS=14** (**000Eh**). Il valore **14** indica quindi che lo stack in quel preciso istante e' in grado di accogliere **14** byte di dati temporanei; rispetto al caso di Figura 1, si tratta di una differenza puramente formale in quanto la sostanza e' sempre la stessa. Possiamo dire quindi che in Figura 2 il fondo dello stack si trova all'indirizzo fisico piu' alto **0BF2Dh**; l'imboccatura dello stack si trova invece all'indirizzo fisico piu' basso **0BF20h**.

Analizziamo ora il funzionamento dello stack di Figura 2 in relazione alle istruzioni **PUSH** e **POP** applicate ad operandi di tipo **WORD**; osserviamo innanzi tutto che dall'indirizzo fisico **0BF20h** si ricava l'indirizzo logico normalizzato **0BF2h:0000h**. La **CPU** quindi indirizza lo stack attraverso coppie logiche **Seg:Offset** aventi la componente **Seg** pari a **0BF2h**; l'inizio dello stack coincide in questo caso con l'inizio del segmento di memoria n. **0BF2h**, per cui i **14** byte dello stack si trovano a spiazzamenti compresi tra **0** e **13** (cioe' tra **0000h** e **000Dh**).

Consideriamo ora una locazione di memoria **A** da **16** bit, e vediamo quello che succede in seguito all'istruzione:

```
PUSH A
```
Questa istruzione comporta due fasi distinte:

1) Il **TOS** viene decrementato di **2** e si ottiene **TOS=12** (cioe' **TOS=000Ch**).
2) La **WORD** presente nella locazione **A** viene letta e inserita all'indirizzo logico **0BF2h:000Ch**, corrispondente all'indirizzo fisico **0BF2Ch**.

Osserviamo che la **WORD** appena inserita nello stack e' formata da due **BYTE** che vanno a disporsi agli offset **000Ch** e **000Dh**; nel rispetto della convenzione **little endian**, il **BYTE** meno significativo occupa l'offset **000Ch**, mentre il **BYTE** piu' significativo occupa l'offset **000Dh**. E' anche importante osservare che la **WORD** appena inserita si trova all'indirizzo fisico **0BF2Ch** che e' pari; cio' dimostra che tutti i dati di tipo **WORD** presenti nello stack vengono allineati in modo ottimale.

Consideriamo ora una locazione di memoria **B** da **16** bit, e vediamo quello che succede in seguito all'istruzione:
```
POP B
```
Questa istruzione comporta due fasi distinte:

1) La **WORD** presente nello stack all'indirizzo logico **0BF2h:000Ch**, viene letta e inserita nella locazione di memoria **B**.
2) Il **TOS** viene incrementato di **2** e si ottiene **TOS=14** (cioe' **TOS=000Eh**).

Osserviamo che l'istruzione **POP** dopo aver estratto la **WORD** che si trova all'indirizzo fisico **0BF2Ch**, non ha bisogno di cancellare il contenuto di questa locazione; infatti, una successiva istruzione **PUSH** sovrascrive direttamente con una nuova **WORD** il vecchio contenuto di quest'area dello stack.

Il funzionamento appena illustrato delle istruzioni **PUSH** e **POP** ci permette di dedurre una serie di importanti aspetti relativi allo stack di un programma; l'aspetto piu' importante e' legato al fatto che, come si puo' facilmente intuire, il programmatore ha il delicato compito di provvedere alla corretta gestione dello stack. Infatti, a differenza di quanto accade con lo stack di Figura 1, nel caso dello stack di un programma nessuno ci impedisce di utilizzare in modo indiscriminato le istruzioni **PUSH** e **POP**; in questo modo e' facile andare incontro a situazioni anomale.
Consideriamo nuovamente lo stack di Figura 2 e supponiamo di partire dalla condizione **TOS=14** (stack vuoto); supponiamo poi che il programma in esecuzione ad un certo punto preveda **4** istruzioni **PUSH** consecutive. Queste **4** istruzioni inseriscono **4 WORD** nello stack portando il **TOS** al valore **6**; se in seguito lo stesso programma prevede **4** istruzioni **POP** consecutive, vengono estratte dallo stack **4 WORD** con il **TOS** che si riporta a **14** (**000Eh**). A questo punto nessuno ci impedisce di ricorrere ad una quinta istruzione **POP** che estrae da **0BF2h:000Eh** una **WORD** che si trova chiaramente al di fuori dello stack; inoltre la stessa istruzione **POP** porta il **TOS** all'offset **16**!
La situazione e' ancora piu' grave nel caso in cui venga eseguita una istruzione **PUSH** con lo stack completamente pieno (**TOS=0**); in questo caso infatti l'istruzione **PUSH** cerca di sottrarre **2** al **TOS** che pero' vale **0**. Siccome il **TOS** e' un valore a **16** bit, si ottiene in complemento a **2**:
```
0 - 2 = 0 + (65536 - 2) = 0 + 65534 = 65534 = FFFEh
```
(Si puo' ricorrere anche all'esempio del contachilometri a **16** bit, andando in retromarcia di **2** bit a partire da **0000000000000000b**).
Si tratta del classico **wrap around** che porta **PUSH** ad inserire una **WORD** all'indirizzo logico **0BF2h:FFFEh** totalmente al di fuori dello stack; trattandosi pero' in questo caso di una operazione di scrittura, viene sovrascritta un'area della memoria che potrebbe anche essere riservata al **SO**, con conseguente crash del programma!
In definitiva, il programmatore e' tenuto a garantire nei propri programmi una gestione rigorosa

dello stack, con un perfetto bilanciamento tra inserimenti ed estrazioni, e con il rispetto dei limiti inferiore e superiore del **TOS**; come vedremo nei capitoli successivi, con un minimo di stile e di attenzione il programmatore puo' svolgere questo compito in modo relativamente semplice.

Cosa succede se nello stack di Figura 2 vogliamo inserire un dato di tipo **BYTE**?
Le **CPU** con architettura a **16** bit o superiore non permettono l'utilizzo delle istruzioni **PUSH** e **POP** con operandi di tipo **BYTE**; in questo modo si evita che il **TOS** si disallinei posizionandosi ad indirizzi dispari.
Se vogliamo inserire nello stack un dato di tipo **BYTE** siamo costretti quindi a trasformare questo **BYTE** in una **WORD**; in sostanza, utilizziamo come operando sorgente di **PUSH** una locazione da **16** bit che contiene nei suoi **8** bit meno significativi il nostro dato di tipo **BYTE**. Al momento di estrarre questo dato dallo stack, utilizziamo l'istruzione **POP** con un operando destinazione ugualmente di tipo **WORD**; il nostro dato di tipo **BYTE** verra' cosi' posizionato negli **8** bit meno significativi dell'operando destinazione.
La situazione e' ancora piu' semplice nel caso in cui nello stack di Figura 2 si voglia memorizzare un dato di tipo **DWORD**; in questo caso infatti ci basta suddividere la **DWORD** in due **WORD** utilizzando poi due istruzioni **PUSH** consecutive. E' fondamentale che il programmatore rispetti la convenzione **little endian** inserendo per prima la **WORD** piu' significativa; in questo modo la **WORD** meno significativa si trova a precedere nello stack la **WORD** piu' significativa.

All'interno dello stack di un programma i dati temporanei vengono continuamente creati e distrutti; la situazione interna dello stack e' in continua evoluzione, per cui e' molto probabile che uno stesso dato temporaneo creato e distrutto numerose volte nello stack, venga posizionato di volta in volta sempre ad offset differenti. Tutto dipende infatti dal valore che assume il **TOS** al momento della creazione di questo dato; la conseguenza pratica di tutto cio' e' data dal fatto che l'indirizzo di un dato temporaneo che si trova nello stack non e' statico ma e' **dinamico**. Per questo motivo, i dati temporanei di un programma vengono anche chiamati **dati dinamici**; e' evidente che un dato dinamico puo' essere utilizzato solo durante la sua esistenza, e cioe' nell'intervallo di tempo che intercorre tra la creazione e la distruzione del dato stesso.
Conoscendo l'indirizzo di un dato dinamico (esistente), possiamo accedere alla corrispondente locazione dello stack modificandone il contenuto; un dato dinamico il cui contenuto puo' essere modificato (variato), viene anche definito **variabile dinamica**.
Il blocco di memoria destinato a contenere i dati dinamici forma un segmento di programma che prende il nome di **Stack Segment** (segmento stack).

Come e' stato gia' detto in precedenza, lo stack di un programma assume una importanza enorme; questo particolare blocco di memoria infatti viene utilizzato non solo dal programma a cui appartiene, ma anche dal **SO** e dalla **CPU**!
Ogni volta che un programma chiama un suo sottoprogramma, la **CPU** salva nello stack il cosiddetto **return address** (indirizzo di ritorno); si tratta dell'indirizzo di memoria da cui riprendera' l'esecuzione al termine del sottoprogramma stesso.
Ogni volta che la **CPU** riceve una richiesta di interruzione hardware, prima di chiamare la relativa **ISR** sospende il programma in esecuzione salvando nello stack tutte le necessarie informazioni; al termine della **ISR** il programma che era stato sospeso viene riavviato grazie alle informazioni estratte dallo stack.
Molti linguaggi di alto livello utilizzano lo stack per creare le cosiddette **variabili locali**; si tratta delle variabili create temporaneamente dai sottoprogrammi che ne hanno bisogno. Nel momento in cui un sottoprogramma termina, le sue variabili locali non sono piu' necessarie e devono essere quindi distrutte; come abbiamo visto in precedenza, lo stack si presta in modo perfetto per questo particolare scopo.
Nei capitoli successivi vedremo che la struttura a stack si presta in modo ottimale anche per la

gestione delle **chiamate innestate** tra sottoprogrammi; si hanno le chiamate innestate quando un sottoprogramma **A** chiama un sottoprogramma **B** che a sua volta chiama un sottoprogramma **C** e cosi' via.
Un caso particolarmente importante di chiamata innestata si ha quando un sottoprogramma **A** chiama se stesso direttamente o indirettamente; in questo caso si parla di **chiamata ricorsiva**. Anche in questo caso vedremo che lo stack permette di gestire in modo semplice ed efficiente la ricorsione.

10.1.3 Segmento misto Data + Stack

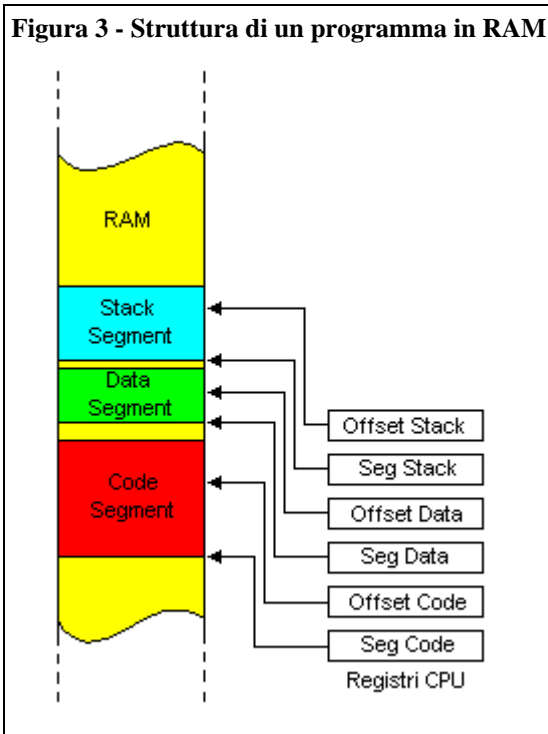
Anziche' dotare un programma di **Data Segment** e **Stack Segment** separati, e' possibile servirsi di un unico segmento di programma destinato a contenere sia i dati statici, sia lo stack; in questo caso, il procedimento da seguire puo' essere facilmente dedotto dalla Figura 2. Prima di tutto dobbiamo allocare staticamente un blocco di memoria sufficiente a contenere sia i dati statici, sia lo stack; a questo punto dobbiamo suddividere questo blocco in due parti destinate una ai dati statici e l'altra ai dati dinamici.
Osservando la Figura 2 si nota che lo stack inizia a riempirsi a partire dagli offset piu' alti; man mano che lo stack si riempie, il **TOS** si decrementa portandosi verso gli offset piu' bassi. E' chiaro quindi che il nostro segmento misto contenente **Data** e **Stack** deve essere organizzato disponendo i dati statici agli offset piu' bassi; gli offset piu' alti vengono riservati invece allo stack.
Supponiamo in Figura 2 di riservare i primi **6** offset ai dati statici; in questo caso il blocco **Data** risulta compreso tra l'offset **0000h** e l'offset **0005h**. Tutti gli offset compresi tra **0006h** e **000Dh** sono riservati invece allo stack; la condizione di stack vuoto e' rappresentata da **TOS=000Eh**, mentre la condizione di stack pieno e' rappresentata da **TOS=0006h**. Se il **TOS** scende sotto l'offset **0006h** invade l'area riservata ai dati statici; in questo caso, eventuali istruzioni **PUSH** sovrascrivono gli stessi dati statici.
Il segmento misto **Data + Stack** viene largamente utilizzato da molti linguaggi di alto livello; questo argomento verra' trattato in un apposito capitolo.

10.1.4 Il Code Segment di un programma

Per la memoria da riservare al blocco codice di un programma si segue lo stesso metodo gia' illustrato per il **Data Segment**; osserviamo infatti che l'insieme dei codici macchina delle varie istruzioni occupa in memoria un ben preciso numero di byte. La cosa migliore da fare consiste quindi nell'allocare staticamente un blocco di memoria le cui dimensioni sono sufficienti a contenere i codici macchina di tutte le istruzioni del programma; il blocco di memoria destinato a contenere le istruzioni forma un segmento di programma che prende il nome di **Code Segment** (segmento codice).

10.1.5 Struttura a segmenti di un programma

A questo punto siamo finalmente in grado di dire che un programma destinato a girare in modalita' reale **8086**, nel caso piu' semplice e intuitivo risulta formato da tre segmenti di programma chiamati **Code Segment**, **Data Segment** e **Stack Segment**; in linea di principio, la dimensione in byte di un programma e' data dalla somma delle dimensioni in byte di questi tre segmenti. Al momento di caricare un programma in memoria per la fase di esecuzione, il **DOS** predispone un blocco di memoria le cui dimensioni sono sufficienti a contenere il programma stesso; la Figura 3 illustra la situazione che si viene a creare quando il nostro programma viene caricato nella **RAM**.

Figura 3 - Struttura di un programma in RAM

Teoricamente i tre segmenti di programma dovrebbero essere consecutivi e contigui, cioe' attaccati l'uno all'altro; bisogna ricordare pero' che al programmatore viene data la possibilita' di selezionare il tipo di allineamento in memoria per ciascun segmento. Di conseguenza, come vedremo in dettaglio nei capitoli successivi, tra un segmento e l'altro possono rimanere degli spazi vuoti inutilizzati; questi spazi vuoti vengono chiamati in gergo **memory holes** (buchi di memoria).

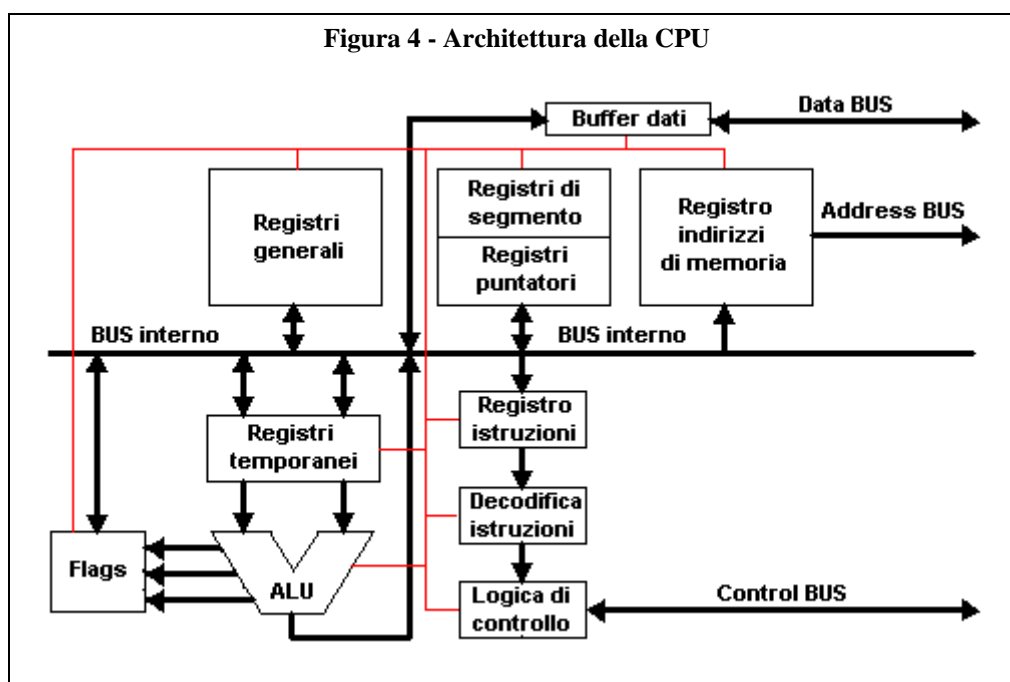
La **CPU** istante per istante deve avere il controllo totale sul programma in esecuzione, e questo significa che deve sapere in quale area della memoria si trovano rispettivamente il blocco dati, il blocco codice e il blocco stack, qual'e' l'indirizzo della prossima istruzione da eseguire, qual'e' l'indirizzo dei dati eventualmente chiamati in causa da questa istruzione, qual'e' l'attuale livello di riempimento dello stack (indirizzo del **TOS**) e cosi' via; come gia' sappiamo, tutte queste informazioni vengono gestite dalla **CPU** attraverso i propri registri interni.
In un precedente capitolo e' stato anche detto che tutti i registri della **CPU** destinati a contenere una componente **Seg** vengono chiamati **Segment Registers** (registri di segmento); tutti i registri della **CPU** destinati a contenere una componente **Offset** vengono chiamati **Pointer Registers** (registri puntatori).
Osservando la Figura 3 possiamo subito dire che una **CPU** della famiglia **80x86** deve essere dotata come minimo di tre registri di segmento destinati a contenere le componenti **Seg** del **Code Segment**, del **Data Segment** e dello **Stack Segment**; in Figura 3 queste tre componenti sono state chiamate **Seg Code**, **Seg Data** e **Seg Stack**.
In relazione al **Code Segment** osserviamo subito che le istruzioni vengono eseguite dalla **CPU** una alla volta; questo significa che per gestire la componente **Offset** dell'indirizzo della prossima istruzione da eseguire e' necessario un unico registro puntatore appositamente destinato a questo scopo.
Nel caso invece dello **Stack Segment** e soprattutto del **Data Segment**, sarebbe opportuno avere a disposizione il maggior numero possibile di registri puntatori; e' chiaro infatti che e' molto meglio poter indirizzare **n** dati con **n** registri puntatori differenti, piuttosto che **n** dati con un solo registro puntatore da far "puntare" di volta in volta su uno dei dati stessi.
E' anche importante che la **CPU** ci metta a disposizione un adeguato numero di registri di uso generale che possiamo utilizzare per contenere ad esempio gli operandi di una operazione logico

aritmetica; come gia' sappiamo infatti, i registri della **CPU** sono accessibili molto piu' velocemente rispetto alle locazioni di memoria.

Sulla base di queste considerazioni, analizziamo ora l'architettura interna di una generica **CPU** appartenente alla famiglia **80x86**.

10.2 Architettura generale della CPU

La struttura interna di una **CPU** varia enormemente da modello a modello; in generale pero' alcune parti sono sempre presenti e questo permette di rappresentare una generica **CPU** secondo lo schema a blocchi di Figura 4.

Figura 4 - Architettura della CPU

Il primo aspetto da sottolineare riguarda il fatto che lo scambio di dati tra i vari blocchi della **CPU** avviene attraverso un apposito bus chiamato **Internal Bus** (bus interno), dotato di un numero di linee che nel caso generale rispecchia l'architettura della stessa **CPU** (**8** bit, **16** bit, **32** bit, etc); in alcuni rari casi puo' capitare che il bus interno abbia un numero di linee maggiore di quello del **Data Bus** del computer (questo accadeva ad esempio nel caso della **CPU 8088** che aveva un bus interno a **16** bit e un **Data Bus** a **8** bit).
Tornando alla Figura 4, possiamo suddividere questo schema in due parti fondamentali: la meta' a sinistra rappresenta la **Execution Unit** o **EU** (unita' di esecuzione), mentre la meta' a destra rappresenta la **Bus Interface Unit** o **BIU** (unita' di interfacciamento con i bus di sistema).

10.2.1 Execution Unit

La **EU**, come si intuisce dal nome, ha il compito di eseguire materialmente le istruzioni del programma in esecuzione ed e' incentrata sul dispositivo che in Figura 4 e' stato chiamato **ALU** o **Arithmetic Logic Unit** (unita' logico aritmetica); la **ALU** contiene i circuiti logici che abbiamo visto nei capitoli precedenti e che sono necessari per eseguire le operazioni logico aritmetiche sui dati del programma. Tecnologie sempre piu' sofisticate permettono di inserire in spazi ristretti un numero sempre maggiore di porte logiche, e questo consente di realizzare delle **ALU** capaci di eseguire un insieme di operazioni sempre piu' vasto e complesso; nel caso generale comunque, qualsiasi **ALU** mette a disposizione una serie di operazioni basilari come ad esempio: addizione,

sottrazione, moltiplicazione, divisione, shift a destra e a sinistra, complementazione dei bit, operatori logici **AND**, **OR**, **EX-OR** e cosi' via.
Naturalmente la **ALU** deve essere in grado di operare su dati la cui dimensione in bit rispecchia l'architettura della **CPU**; possiamo dire quindi che su una **CPU** con architettura a **16** bit, la **ALU** deve essere in grado di operare su dati a **16** bit, cosi' come su una **CPU** con architettura a **32** bit, la **ALU** deve essere in grado di operare su dati a **32** bit.

La **ALU** non e' dotata di memoria propria per cui deve servirsi di apposite memorie esterne che in Figura 4 sono rappresentate dai blocchi denominati: **Registri temporanei**, **Registri generali** e **Flags** (registro dei flags).
I registri temporanei non sono accessibili al programmatore, ed hanno il compito di memorizzare temporaneamente i dati (**operandi**) sui quali la **ALU** deve eseguire una determinata operazione logico aritmetica; ogni volta che arrivano nuovi dati da elaborare, il vecchio contenuto dei registri temporanei viene sovrascritto.
I registri generali sono invece direttamente accessibili al programmatore e rappresentano una vera e propria memoria **RAM** interna velocissima, che viene utilizzata per svolgere svariate funzioni; questi registri ad esempio contengono dati in arrivo dalle periferiche (cioe' dalla **RAM** o dai dispositivi di **I/O**), dati destinati alle periferiche, risultati di operazioni appena eseguite dalla **ALU** e cosi' via.
Come si puo' notare in Figura 4, la **ALU** e' collegata ai registri generali sia in ingresso che in uscita e puo' quindi dialogare direttamente con essi; il programmatore **Assembly** puo' utilizzare in modo massiccio i registri generali per permettere alla **ALU** di svolgere svariate operazioni alla massima velocita' possibile. Bisogna ricordare infatti che i registri della **CPU** hanno tempi di accesso nettamente inferiori rispetto alle locazioni della **RAM**; tutte le operazioni che coinvolgono dati presenti nei registri, vengono svolte dalla **ALU** molto piu' velocemente rispetto al caso in cui i dati si trovino invece in **RAM**.
Proprio per questo motivo, persino alcuni linguaggi di programmazione di alto livello danno la possibilita' al programmatore di richiedere l'inserimento di determinati dati del programma, direttamente nei registri della **CPU**; il linguaggio **C** ad esempio, mette a disposizione la parola chiave **register** che applicata ad un dato (intero), indica al compilatore **C** di inserire, se e' possibile, quel dato in un registro libero della **CPU**. Il compilatore **C** e' libero di ignorare eventualmente questa richiesta; il vantaggio dell'**Assembly** sta proprio nel fatto che in questo caso il programmatore ha l'accesso diretto ai registri generali con la possibilita' quindi di sfruttarli al massimo.

Un altro importante componente della **EU** e' il **Registro dei Flags**, che istante per istante contiene una serie di informazioni che permettono, sia al programmatore che alla **CPU** di avere un quadro completo e dettagliato sul risultato prodotto da una operazione appena eseguita dalla **ALU**; in base a queste informazioni, il programmatore puo' prendere le decisioni piu' opportune per determinare il cosiddetto **flusso del programma**, cioe' l'ordine di esecuzione delle varie istruzioni.
Abbiamo gia' fatto conoscenza nei precedenti capitoli con il **Carry Flag**, il **Sign Flag** e l'**Overflow Flag**; piu' avanti verranno illustrati altri flags, compresi quelli che hanno lo scopo di configurare la modalita' operativa della **CPU**. L'insieme dei valori assunti da questi flags definisce la cosiddetta **Program Status Word** o **PSW** (parola di stato del computer).

10.2.2 Bus Interface Unit

Passiamo ora alla meta' destra dello schema di Figura 4, che rappresenta la **BIU** ed ha il compito importantissimo di far comunicare la **CPU** con il mondo esterno attraverso i bus di sistema; come si puo' notare, anche la **BIU** comprende numerosi registri di memoria inclusi nei blocchi denominati **Registri di segmento** e **Registri puntatori**. Come abbiamo gia' visto, attraverso questi registri la

CPU controlla in modo completo l'indirizzamento del programma in esecuzione; si tratta infatti dei registri destinati a contenere gli indirizzi relativi ai vari segmenti che formano un programma.
Tra i vari registri puntatori e' necessario segnalare in particolare il cosiddetto **PC** o **Program Counter** (contatore di programma), che contiene l'indirizzo della prossima istruzione da eseguire; nel momento in cui la **CPU** sta eseguendo una istruzione, il **PC** viene aggiornato in modo che punti alla prossima istruzione da eseguire. Nelle **CPU** della famiglia **80x86** il **PC** viene chiamato **IP** o **Instruction Pointer** (puntatore alla prossima istruzione da eseguire); questi argomenti vengono trattati in dettaglio piu' avanti.

Il blocco che in Figura 4 viene chiamato **Control Logic** o **CL** (logica di controllo) rappresenta come gia' sappiamo il cervello di tutto il sistema in quanto ha il compito di gestire tutto il funzionamento della **CPU**; la **CL** stabilisce istante per istante, quale dei dispositivi mostrati in Figura 4 deve ricevere il controllo, che compito deve eseguire questo dispositivo, quali dispositivi devono essere invece inibiti, etc.
In Figura 4, le linee di colore rosso rappresentano simbolicamente i collegamenti che permettono alla **CL** di pilotare tutti i dispositivi della **CPU**; attraverso poi il **Control Bus**, la **CL** puo' anche interagire con i dispositivi esterni alla **CPU** (**RAM**, dispositivi di **I/O**, etc).

Per avere un'idea dell'importanza della **CL**, analizziamo in modo semplificato quello che accade quando deve essere eseguita una istruzione del tipo:

```
NOT Variabile1
```

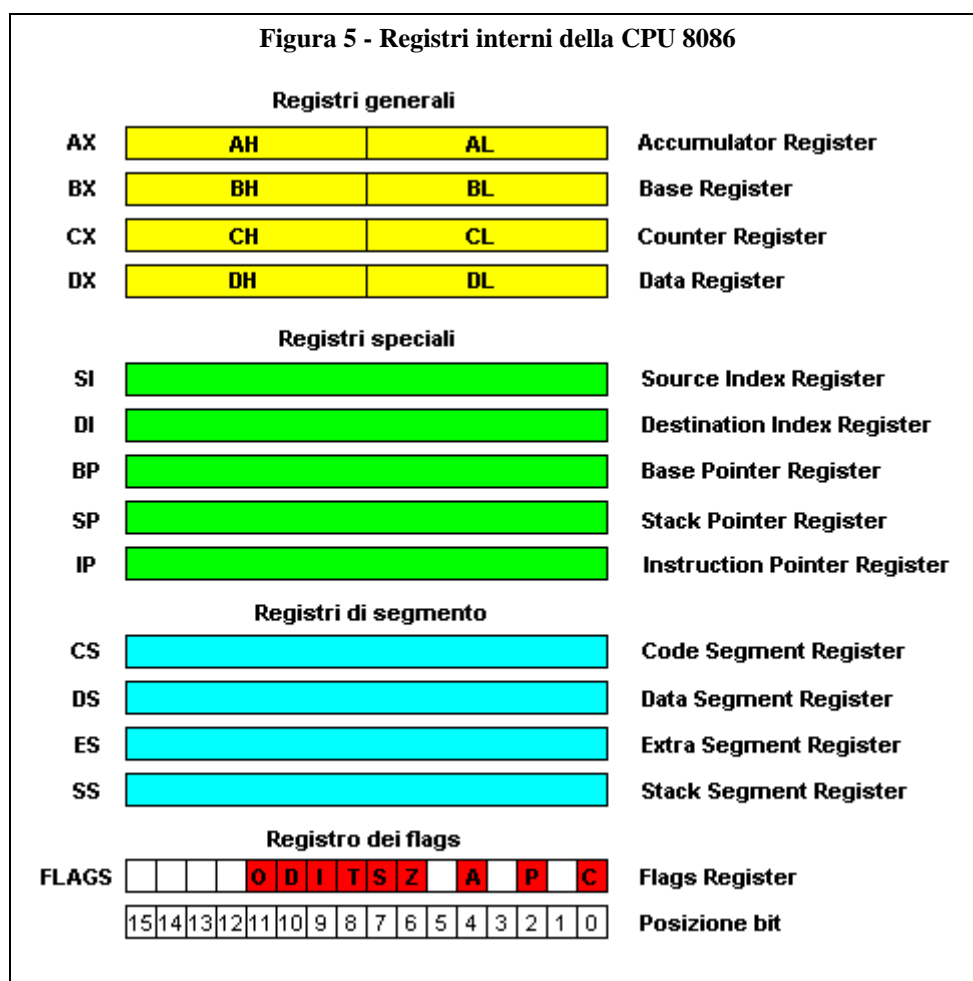
Il nome **Variabile1** rappresenta simbolicamente una locazione di memoria che contiene ad esempio un numero intero a **16** bit; la precedente istruzione deve quindi invertire tutti i **16** bit contenuti nella locazione **Variabile1**. Come vedremo nei prossimi capitoli, dal punto di vista della **CPU** il simbolo **Variabile1** rappresenta l'indirizzo di memoria della locazione contenente il dato a cui vogliamo accedere; tenendo conto di questo aspetto, l'esecuzione della precedente istruzione comporta le seguenti fasi:

***** La **CL** legge l'indirizzo **Seg:Offset** relativo alla prossima istruzione da eseguire, e lo passa al dispositivo che in Figura 4 viene chiamato **Registro indirizzi di memoria** o **Memory Address Register** (**MAR**).
***** Il **MAR** ha il compito (in modalita' reale) di convertire la coppia **Seg:Offset** in un indirizzo fisico a **20** bit da caricare sull'**Address Bus**.
***** La **CL** attiva l'**Address Bus** e mette in comunicazione la **CPU** con la locazione del **Code Segment** che contiene il codice macchina della prossima istruzione da eseguire.
***** La **CL** attiva il **Data Bus** in lettura in modo che il codice macchina venga letto dalla memoria e caricato nel dispositivo che in Figura 4 viene chiamato **Registro Istruzioni**.
***** Il codice macchina viene decodificato dal dispositivo che in Figura 4 viene chiamato **Decodifica Istruzioni**, e viene accertata in questo modo la richiesta di eseguire una istruzione **NOT** sui **16** bit del dato **Variabile1**.
***** La **CL** legge l'indirizzo **Seg:Offset** relativo a **Variabile1** e lo passa al **MAR**.
***** Il **MAR** converte la coppia **Seg:Offset** in un indirizzo fisico a **20** bit da caricare sull'**Address Bus**.
***** La **CL** attiva l'**Address Bus** e mette in comunicazione la **CPU** con la locazione del **Data Segment** che contiene **Variabile1**.
***** La **CL** attiva il **Data Bus** in lettura permettendo il trasferimento del contenuto di **Variabile1** in uno dei registri temporanei della **ALU**.
***** La **CL** attiva sulla **ALU** la rete logica che esegue l'operazione **NOT** sul contenuto del registro temporaneo selezionato.
***** La **CL** attiva il **Data Bus** in scrittura permettendo il trasferimento del contenuto del registro temporaneo nella locazione **Variabile1**.

Dopo aver esaminato la struttura generale di una ipotetica **CPU**, vediamo come questi concetti vengono applicati ai microprocessori realmente esistenti della famiglia **80x86**; naturalmente, ci interessa analizzare in dettaglio le caratteristiche dei registri, e cioe' di quelle parti della **CPU** che sono direttamente accessibili al programmatore.

10.3 Registri interni delle CPU a 16 bit

Nella famiglia **80x86** la **CPU** di riferimento per le architetture a **16** bit e' sicuramente l'**8086**; la Figura 5 illustra il set completo di registri interni (tutti a **16** bit) di questa **CPU**.

Figura 5 - Registri interni della CPU 8086

Osserviamo subito che l'**8086** dispone di **4** registri di segmento che sono **CS**, **DS**, **ES** e **SS**; come si deduce facilmente dai nomi:

CS e' destinato a contenere la componente **Seg** del **Code Segment**;
DS e' destinato a contenere la componente **Seg** del **Data Segment**;
SS e' destinato a contenere la componente **Seg** dello **Stack Segment**.

Il registro **ES** permette di gestire la componente **Seg** di un secondo **Data Segment**; possiamo dire quindi che con l'**8086**, il programmatore ha eventualmente la possibilita' di gestire contemporaneamente due **Data Segment** distinti. La presenza di **ES** e' necessaria in quanto l'**8086** dispone di istruzioni che, come vedremo in un apposito capitolo, permettono di trasferire dati tra due blocchi di memoria; il comportamento predefinito di queste istruzioni prevede che il blocco

sorgente venga gestito con **DS**, e il blocco destinazione con **ES**.

I registri che in Figura 5 vengono definiti **speciali**, sono tutti registri puntatori; il loro scopo quindi e' quello di gestire componenti **Offset** relative ai vari segmenti di programma.

Il registro **IP** fa' coppia con **CS**, in quanto e' destinato a contenere la componente **Offset** dell'indirizzo della prossima istruzione da eseguire; questo registro non e' accessibile in modo diretto al programmatore.
Il compito di inizializzare la coppia **CS:IP** spetta rigorosamente al **linker**; a tale proposito, il programmatore e' tenuto a specificare obbligatoriamente in ogni programma, l'indirizzo della prima istruzione da eseguire. Questo indirizzo prende il nome di **entry point** (punto d'ingresso) e permette appunto al **linker** di determinare la coppia iniziale **Seg:Offset** che verra' caricata in **CS:IP**.
In fase di esecuzione di un programma, la gestione di **CS:IP** passa alla **CPU** che provvede ad aggiornare istante per istante **IP** (e se necessario anche **CS**); osserviamo, infatti, che l'ordine con il quale verranno eseguite le varie istruzioni del nostro programma e' implicito nella struttura del programma stesso.
Il programmatore ha la possibilita' di accedere indirettamente a **CS** e a **IP**, modificandone il contenuto; se si eccettuano casi rarissimi (e molto complessi), il programmatore deve evitare nella maniera piu' assoluta qualsiasi modifica alla coppia **CS:IP**!

I registri puntatori **SP** e **BP** fanno coppia normalmente con **SS** e sono quindi destinati a contenere componenti **Offset** relative allo **Stack Segment** del programma; in particolare, istante per istante **SP** rappresenta il **TOS** dello stack. L'utilizzo di **SP** e' riservato quindi principalmente al **SO** e alla **CPU**; in alcuni rari casi, **SP** deve essere gestito direttamente anche dal programmatore.
Il registro **BP** puo' essere, invece, utilizzato dal programmatore per muoversi liberamente all'interno dello stack; lo scopo fondamentale di **BP** e' quello di permettere al programmatore di accedere rapidamente ai dati temporanei eventualmente salvati nello stack.
BP viene utilizzato per lo stesso scopo anche da molti compilatori dei linguaggi di alto livello; in questo caso, **BP** ha il compito di indirizzare, ad esempio, le variabili locali e la lista dei parametri dei sottoprogrammi. Queste informazioni formano il cosiddetto **stack frame** di un sottoprogramma; in un apposito capitolo, questi argomenti verranno trattati in modo dettagliato.
L'inizializzazione della coppia **SS:SP** puo' essere effettuata in modo automatico dal **linker**; in alternativa, si puo' anche optare per una inizializzazione manuale a carico del programmatore. Per delegare al **linker** il compito di inizializzare in modo automatico la coppia **SS:SP**, dobbiamo creare un segmento di programma esplicitamente "etichettato" come **Stack Segment**; in questo caso il **linker** carica in **SS** la componente **Seg** dell'indirizzo iniziale del blocco stack, e in **SP** la componente **Offset** del **TOS** iniziale (che, come gia' sappiamo, corrisponde alla massima capienza in byte dello stack). In assenza di uno **Stack Segment** esplicito, l'inizializzazione di **SS:SP** deve essere effettuata rigorosamente dal programmatore; questi argomenti verranno analizzati in dettaglio nei prossimi capitoli.

I registri puntatori **SI** e **DI** fanno coppia normalmente con **DS** e sono quindi destinati a contenere componenti **Offset** relative al **Data Segment** del programma; il programmatore puo' quindi indirizzare i dati di un programma attraverso coppie del tipo **DS:SI** e **DS:DI**. L'inizializzazione di **DS** spetta unicamente al programmatore; e' chiaro infatti che e' compito del programmatore indicare alla **CPU** l'indirizzo dei dati ai quali si vuole accedere.
I due registri **SI** e **DI** vengono anche utilizzati automaticamente dalla **CPU** con le istruzioni citate in precedenza per il trasferimento dati tra due blocchi di memoria; in questo caso la **CPU** utilizza **DS:SI** come sorgente e **ES:DI** come destinazione. Cio' spiega anche il perche' dei nomi assegnati a **SI** e **DI**; infatti, la sigla **SI** sta per **indice sorgente**, mentre la sigla **DI** sta per **indice destinazione**.
Una caratteristica particolare di **SI** e **DI** e' data dal fatto che questi due registri possono essere

utilizzati, non solo come registri puntatori, ma anche come registri generali; il programmatore e' libero quindi di utilizzare **SI** e **DI** per gestire informazioni di vario genere.

Passiamo infine al gruppo dei **4** registri generali chiamati **AX**, **BX**, **CX** e **DX**; trattandosi di registri generali, il programmatore li puo' utilizzare liberamente per svariati scopi. In particolare, i registri generali vengono impiegati in modo veramente massiccio per contenere dati che devono essere velocemente elaborati dalla **ALU**; i nomi di questi **4** registri sono strettamente legati al ruolo predefinito che essi svolgono nella **CPU 8086**.
Il registro **AX** viene chiamato **accumulatore** e svolge il ruolo di registro preferenziale della **CPU**; in generale, tutte le famiglie di **CPU** hanno almeno un registro accumulatore. La peculiarita' dell'accumulatore sta nel fatto che esso viene largamente utilizzato dalla **CPU** come registro predefinito per numerose operazioni; nel caso dell'**8086**, il registro **AX** viene utilizzato, ad esempio, come operando predefinito per moltiplicazioni, divisioni, comparazioni, etc. Un altro impiego predefinito di **AX** e' quello di registro destinazione o sorgente per lo scambio di dati con le periferiche di **I/O**; appare evidente il fatto che per il programmatore, un utilizzo occulato di **AX** puo' portare ad ottenere un notevole miglioramento delle prestazioni di un programma.
Il registro **BX** ha caratteristiche del tutto simili a **SI** e **DI**; infatti, anche **BX** puo' essere usato sia come registro generale, sia come registro puntatore. Il suo nome (registro base) deriva proprio dall'impiego come puntatore in combinazione con **SI** e **DI**; come vedremo in un apposito capitolo, attraverso questi registri e' possibile specificare componenti **Offset** del tipo **BX+DI**. In una componente **Offset** di questo genere il registro **BX** viene appunto chiamato **base**, mentre **DI** viene chiamato **indice**; con questo sistema e' possibile indirizzare in modo molto efficiente, vettori, matrici e altre strutture dati. Il registro **BX** impiegato come puntatore, normalmente fa' coppia con **DS**.
Il registro **CX** viene chiamato **contatore** in quanto la **CPU** lo utilizza in diverse istruzioni come registro predefinito per effettuare dei conteggi; naturalmente, come accade per tutti i registri generali, il programmatore e' libero di utilizzare **CX** anche per altri scopi.
Il registro **DX**, infine, viene chiamato **registro dati** in quanto il suo ruolo predefinito e' quello di contenere dati da impiegare come operandi in varie operazioni logico aritmetiche; questo registro viene anche impiegato per contenere l'indirizzo di una porta hardware che deve comunicare con la **CPU**.

Osserviamo in Figura 5 che i **4** registri generali **AX**, **BX**, **CX** e **DX** sono tutti a **16** bit; al programmatore viene data anche la possibilita' di suddividere ciascuno di essi in due **Half Registers** (mezzi registri) da **8** bit. A tale proposito, basta sostituire la lettera **X** con la lettera **H** o **L**; la **H** sta per **High** (alto), mentre la **L** sta per **Low** (basso). Nel caso, ad esempio, di **AX**, la sigla **AH** sta per **Accumulator High**, e indica gli **8** bit piu' significativi di **AX**; la sigla **AL** sta invece per **Accumulator Low**, e indica gli **8** bit meno significativi di **AX**.

10.3.1 Associazioni predefinite tra registri puntatori e registri di segmento

Le considerazioni appena esposte, assumono una importanza enorme nella scrittura di un programma **Assembly**; e' necessario quindi analizzare in dettaglio gli aspetti relativi alle associazioni predefinite che la **CPU** effettua tra registri di segmento e registri puntatori.

Partiamo quindi dalle convenzioni fondamentali seguite dalla **CPU**:

CS e' il registro di segmento naturale per il **Code Segment** di un programma.

DS e' il registro di segmento naturale per il **Data Segment** di un programma.

SS e' il registro di segmento naturale per lo **Stack Segment** di un programma.

Sulla base di queste convenzioni, la **CPU** stabilisce una serie di importanti associazioni predefinite tra registri puntatori e registri di segmento; in assenza di diverse indicazioni da parte del programmatore, la **CPU** adotta il seguente comportamento predefinito:

IP viene automaticamente associato a **CS**

SP e **BP** vengono automaticamente associati a **SS**

SI, **DI** e **BX** vengono automaticamente associati a **DS**

Come e' stato detto in precedenza, la coppia **CS:IP** rappresenta, istante per istante, l'indirizzo logico della prossima istruzione da eseguire; il programmatore, non ha alcuna possibilita' di alterare l'associazione predefinita tra **CS** e **IP**. Del resto, abbiamo visto che la gestione della coppia **CS:IP**, spetta rigorosamente alla **CPU**; e' necessario evitare nella maniera piu' assoluta, di accedere indirettamente a questi registri, modificandone il contenuto.

Il programmatore ha invece la possibilita' di gestire come meglio crede, le associazioni predefinite che coinvolgono **SP**, **BP**, **SI**, **DI** e **BX**; in particolare, accettando queste associazioni predefinite, possiamo gestire un indirizzo logico attraverso la sola componente **Offset**!
Questa possibilita' e' legata, ovviamente, al fatto che la **CPU** e' in grado, appunto, di associare automaticamente un determinato registro puntatore, all'opportuno registro di segmento (che contiene la componente **Seg** di un indirizzo logico).
Supponiamo di specificare un indirizzo logico attraverso il solo registro **SP**; in base alle convenzioni enunciate in precedenza, la **CPU** associa automaticamente **SP** a **SS**, e ottiene l'indirizzo logico completo **SS:SP**.
Analogamente, supponiamo di specificare un indirizzo logico attraverso il solo registro **DI**; in base alle convenzioni enunciate in precedenza, la **CPU** associa automaticamente **DI** a **DS**, e ottiene l'indirizzo logico completo **DS:DI**.

Un indirizzo logico formato dalla sola componente **Offset**, viene chiamato **indirizzo NEAR** (indirizzo vicino); gli indirizzi **NEAR** sono formati da soli **16** bit, per cui oltre a ridurre le dimensioni del programma, sono anche molto piu' veloci da gestire rispetto agli indirizzi logici completi espressi da coppie **Seg:Offset**.

Se il programmatore vuole alterare le associazioni predefinite tra registri puntatori e registri di segmento, deve indicare in modo esplicito alla **CPU**, a quale registro di segmento (non naturale) vuole associare un determinato registro puntatore.
Se, ad esempio, vogliamo associare **BX** a **CS**, dobbiamo scrivere esplicitamente **CS:BX**; analogamente, se vogliamo associare **BP** a **DS**, dobbiamo scrivere esplicitamente **DS:BP**.

L'alterazione delle associazioni predefinite tra registri di segmento e registri puntatori, prende il nome di **segment override**; in inglese, il verbo **to override** significa, scavalcare, aggirare le regole.
Un indirizzo logico formato da una coppia completa **Seg:Offset**, viene chiamato **indirizzo FAR** (indirizzo lontano); gli indirizzi **FAR** occupano piu' memoria rispetto agli indirizzi **NEAR**, e vengono gestiti meno velocemente dalla **CPU**.
Nei limiti del possibile, e' meglio quindi servirsi sempre di indirizzi di tipo **NEAR**; come vedremo pero' nel seguito del capitolo e nei capitoli successivi, in molte situazioni non esiste alternativa all'utilizzo degli indirizzi **FAR**.

10.3.2 Indirezione o deriferimento di un puntatore NEAR

Supponiamo di avere un dato a **16** bit che si trova all'offset **002Ch** di un **Data Segment** avente componente **Seg** pari a **03FCh**; supponiamo anche che questo dato sia un numero intero espresso dal valore esadecimale **288Ah**.
Se vogliamo gestire questo **Data Segment** con **DS**, dobbiamo porre, ovviamente, **DS=03FCh**; sfruttando ora le associazioni predefinite che legano **DS** a **DI**, **SI** e **BX**, possiamo utilizzare uno di questi registri puntatori, per gestire il nostro dato attraverso un indirizzo di tipo **NEAR**.
Caricando, ad esempio, in **BX** l'offset **002Ch**, possiamo dire che il dato a cui vogliamo accedere, si trova all'indirizzo **NEAR** espresso da **BX**; la **CPU** associa automaticamente **BX** a **DS**, e ottiene l'indirizzo logico completo **DS:BX=03FCh:002Ch**.

Se ora vogliamo accedere a questo dato tramite il suo puntatore, dobbiamo compiere una operazione che prende il nome di **deriferimento del puntatore** (o **indirezione del puntatore**); si dice anche che il puntatore viene **dereferenziato**.
Il linguaggio **Assembly** fornisce una apposita sintassi che ci permette di dereferenziare un puntatore; se **BX** e' un puntatore **NEAR**, allora il simbolo **[BX]** rappresenta il contenuto della locazione di memoria puntata da **BX**. In sostanza:

Simbolo	Contenuto
BX	002Ch
[BX]	288Ah

Se chiediamo alla **CPU** di trasferire nel registro accumulatore **AX** il contenuto di **BX**, otteniamo **AX=002Ch**; in questo caso infatti, la **CPU** legge il valore **002Ch** contenuto in **BX**, e lo copia in **AX**.
Se, invece, chiediamo alla **CPU** di trasferire nel registro accumulatore **AX** il contenuto di **[BX]**, otteniamo **AX=288Ah**; in questo caso infatti, la **CPU** accede all'indirizzo di memoria **DS:BX=03FCh:002Ch**, legge il contenuto **288Ah**, e lo copia in **AX**.

10.3.3 Indirezione o deriferimento di un puntatore FAR
Supponiamo di avere un dato a **16** bit che si trova all'offset **001Ah** di un **Code Segment** avente componente **Seg** pari a **0DF8h**; supponiamo anche che questo dato sia un numero intero espresso dal valore esadecimale **918Fh**.
Quando la **CPU** sta eseguendo le istruzioni presenti in questo **Code Segment**, si ha, ovviamente, **CS=0DF8h**; infatti, sappiamo che la **CPU** utilizza **CS:IP** per accedere alle istruzioni da eseguire.
In una situazione di questo genere, siamo costretti a gestire il nostro dato attraverso un indirizzo di tipo **FAR**; infatti, il registro di segmento **CS**, non e' quello naturale per i dati.
Caricando, ad esempio, in **DI** l'offset **001Ah**, possiamo dire che il dato a cui vogliamo accedere, si trova all'indirizzo **FAR** espresso da **CS:DI=0DF8h:001Ah**; in assenza del segment override esplicito, la **CPU** assocerebbe automaticamente **DI** a **DS**.
In base a quanto e' stato esposto in precedenza sul deriferimento dei puntatori, possiamo dire che se **CS:DI** e' un puntatore **FAR**, allora il simbolo **CS:[DI]** rappresenta il contenuto della locazione di memoria puntata da **CS:DI**; in sostanza:

Simbolo	Contenuto
CS:DI	0DF8h:001Ah
CS:[DI]	918Fh

Se chiediamo alla **CPU** di trasferire nel registro accumulatore **AX** il contenuto di **DI**, otteniamo **AX=001Ah**; in questo caso infatti, la **CPU** legge il valore **001Ah** contenuto in **DI**, e lo copia in **AX**.
Se chiediamo alla **CPU** di trasferire nel registro accumulatore **AX** il contenuto di **[DI]**, otteniamo in **AX** un valore diverso da **918Fh**; in questo caso infatti, la **CPU** associa automaticamente **DI** a **DS**, e copia in **AX** un valore a **16** bit che si trova all'indirizzo **DS:DI**.
Se, invece, chiediamo alla **CPU** di trasferire nel registro accumulatore **AX** il contenuto di **CS:[DI]**, otteniamo **AX=918Fh**; in questo caso infatti, la **CPU** accede all'indirizzo di memoria **CS:DI=0DF8h:001Ah**, legge il contenuto **918Fh**, e lo copia in **AX**.
Come si puo' facilmente intuire, i concetti appena esposti sui puntatori e sulle associazioni predefinite tra registri, assumono una importanza colossale non solo per chi programma in **Assembly**; un programmatore che non acquisisce una adeguata padronanza di questi concetti, va' incontro ad una elevatissima probabilita' di scrivere programmi contenenti errori piuttosto gravi e subdoli.
Tutti i concetti relativi agli indirizzi di memoria e alla loro indirezione, rappresentano il metodo piu' semplice e naturale che permette alla **CPU** di svolgere il proprio lavoro; si tratta quindi degli stessi concetti che si ritrovano (in modo piu' o meno esplicito) in tutti i linguaggi di programmazione di alto livello. Abbiamo gia' visto ad esempio che nel linguaggio **C**, una definizione del tipo:

```
char near *ptcn;
```

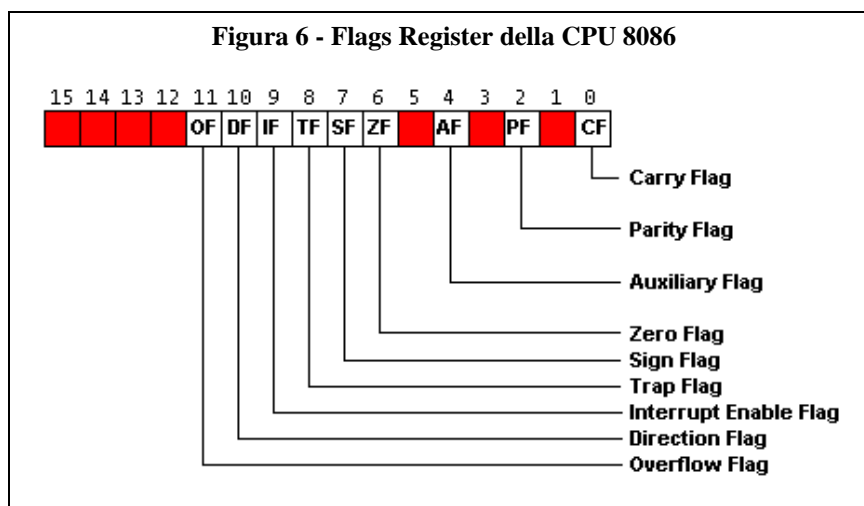
crea un puntatore **NEAR** (intero senza segno a **16** bit) chiamato **ptcn** e destinato a contenere la componente **Offset** dell'indirizzo di un dato di tipo **char** (intero con segno a **8** bit).
Analogamente, la definizione:

```
char far *ptcf;
```

crea un puntatore **FAR** (intero senza segno a **32** bit) chiamato **ptcf** e destinato a contenere la coppia **Seg:Offset** dell'indirizzo di un dato di tipo **char** (intero con segno a **8** bit).
Purtroppo pero', molti libri sulla programmazione con i linguaggi di alto livello, descrivono i puntatori come qualcosa di complesso e misterioso; proprio per questo motivo, capita spesso che i puntatori diventino un vero e proprio incubo per i programmatori. Tanto per citare un esempio emblematico, in un libro sul linguaggio **Pascal**, i puntatori sono stati addirittura definiti "entita' esoteriche"!
Per chi programma in **Assembly**, invece, i puntatori rappresentano il pane quotidiano, e queste paure appaiono assolutamente incomprensibili; abbiamo appena visto, infatti, che si tratta in realta' di concetti abbastanza semplici da capire anche per un programmatore **Assembly** alle prime armi.

10.3.4 Flags Register

La **CPU 8086** dispone anche di un registro a **16** bit chiamato **Flags Register**; come e' stato gia' detto, questo registro, istante per istante, contiene una serie di informazioni che permettono, sia al programmatore, sia alla **CPU**, di avere un quadro completo e dettagliato sul risultato prodotto da una operazione appena eseguita dalla **ALU**.
Il **Flags Register** (o **FLAGS**) e' suddiviso in diversi campi che, in genere, hanno una ampiezza pari ad **1** bit; la Figura 6 illustra la disposizione di questi campi.

Figura 6 - Flags Register della CPU 8086

I bit colorati in rosso sono inutilizzati e in ogni caso, si tratta di campi riservati alle future **CPU**; i manuali della **8086** consigliano di evitare la modifica del contenuto di questi bit.
I bit colorati in bianco rappresentano ciascuno un differente flag (segnalatore); notiamo in particolare la presenza dei flags **CF**, **SF** e **OF** che gia' conosciamo. Nel caso generale, ciascuno di questi bit indica se, dopo lo svolgimento di una operazione logico aritmetica, una determinata condizione si e' verificata o meno; se il bit vale **1**, la condizione si e' verificata, mentre se il bit vale **0**, la condizione non si e' verificata.

Il flag **CF** o **Carry Flag** (segnalatore di riporto/prestito) vale **1** quando si verifica un riporto dopo un'addizione, o quando si verifica un prestito dopo una sottrazione; in caso contrario, **CF** vale zero. Nei prossimi capitoli vedremo altri significati assunti, sia dal flag **CF**, sia dagli altri flags.

Il flag **PF** o **Parity Flag** (segnalatore di parita') viene largamente utilizzato per il controllo degli errori durante la trasmissione di dati tra due computers attraverso dispositivi come il modem; quando la **CPU** esegue determinate istruzioni logico aritmetiche, controlla gli **8** bit meno significativi del risultato prodotto dalle istruzioni stesse, e attraverso **PF** indica se questi **8** bit contengono un numero pari o dispari di **1**. Se il numero di **1** e' pari la **CPU** pone **PF=1**; se il numero di **1** e' dispari la **CPU** pone **PF=0**.
La ragione per cui **PF** si riferisce solo agli **8** bit meno significativi di un numero e' legata al fatto che diversi protocolli di comunicazione tra computers prevedono la suddivisione dei dati trasmessi, in gruppi (pacchetti) di **8** bit; questi protocolli utilizzano per i dati la vecchia codifica **ASCII** a **7** bit che permette di rappresentare 2^7**=128** simboli differenti. I **7** bit meno significativi di ogni pacchetto contengono un codice **ASCII**, mentre il bit piu' significativo contiene il cosiddetto **bit di parita'**; un bit di parita' di valore **1** prende il nome di **parita' pari**, e indica che il pacchetto e' valido solo se i suoi **8** bit hanno un numero pari di **1**. Un bit di parita' di valore **0** prende il nome di **parita' dispari**, e indica che il pacchetto e' valido solo se i suoi **8** bit hanno un numero dispari di **1**; questo sistema si basa sul fatto che se la probabilita' di inviare un bit sbagliato e' bassa, la probabilita' di inviare due bit sbagliati diventa enormemente piu' bassa.

Il flag **AF** o **Auxiliary Flag** (segnalatore ausiliario) vale **1** quando si verifica un riporto dal nibble basso al nibble alto di un numero a **8** bit, o quando si verifica un prestito dal nibble alto al nibble basso di un numero a **8** bit; come vedremo in un apposito capitolo, **AF** viene utilizzato nell'aritmetica dei numeri in formato **BCD**, cioe' in formato **Binary Coded Decimal** (decimale codificato in binario).

Il flag **ZF** o **Zero Flag** (segnalatore di zero) vale **1** quando il risultato di una operazione e' zero; in caso contrario si ha **ZF=0**.

Il flag **SF** o **Sign Flag** (segnalatore di segno) vale **1** quando il risultato di una operazione ha il bit piu' significativo (**MSB**) che vale **1**; in caso contrario si ottiene **SF=0**. Come gia' sappiamo, i numeri interi con segno espressi in complemento a **2** sono negativi se hanno l'**MSB** che vale **1**, mentre sono positivi in caso contrario.

Il flag **OF** o **Overflow Flag** (segnalatore di trabocco) vale **1** quando il risultato di una operazione tra numeri con segno e' "troppo positivo" o "troppo negativo"; in sostanza, **OF=1** segnala un overflow di segno, cioe' un risultato il cui bit di segno e' l'opposto di quello che sarebbe dovuto essere.

I flags appena elencati, vengono modificati direttamente dalla **CPU**; nel **Flags Register** dell'**8086** sono presenti anche altri tre flags che vengono gestiti, invece, dal programmatore, e servono a configurare il modo di operare della **CPU**.

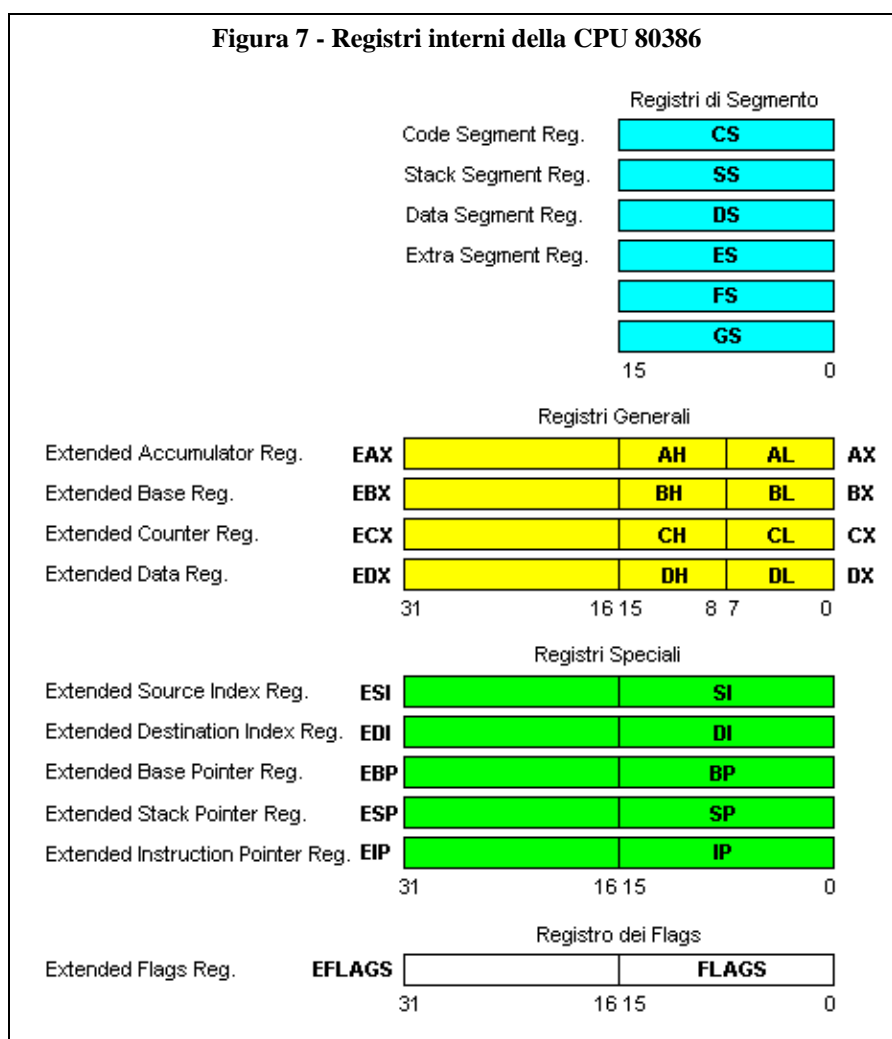
Il flag **TF** o **Trap Flag** (segnalatore di trappola) viene utilizzato da appositi programmi chiamati **debuggers** per scovare eventuali errori in un programma in esecuzione; se **TF** viene posto a **1**, la **CPU** genera una interruzione hardware per ogni istruzione eseguita, e cio' permette ai debuggers (che intercettano l'interruzione) di analizzare, istruzione per istruzione, il programma in esecuzione (si parla in questo caso di **esecuzione a passo singolo**). Se non si deve effettuare nessun lavoro di "debugging", si consiglia vivamente di tenere **TF** a **0** in modo da evitare di rallentare il computer; questo flag viene inizializzato a **0** in fase di accensione del **PC**.

Il flag **IF** o **Interrupt Enable Flag** (segnalatore di interruzioni abilitate) serve ad abilitare o a disabilitare la gestione delle interruzioni hardware che le varie periferiche inviano alla **CPU** per segnalare di voler intervenire; tutte le interruzioni hardware che possono essere abilitate o disabilitate tramite **IF** vengono chiamate **maskable interrupts** (interruzioni mascherabili).
Si puo' avere la necessita' di disabilitare temporaneamente queste interruzioni, quando, ad esempio, un programma sta eseguendo un compito piuttosto delicato che richiede la totale assenza di "interferenze" esterne; ponendo **IF=0** le interruzioni mascherabili vengono disabilitate (mascherate), mentre se **IF=1** le interruzioni mascherabili vengono abilitate ed elaborate dalla **CPU**.
Lo stato di **IF** non ha nessun effetto sulle interruzioni software (che vengono inviate esplicitamente dai programmi) e sulle cosiddette **NMI** o **Non Maskable Interrupts** (interruzioni non mascherabili); le **NMI** vengono inviate dall'hardware alla **CPU** nel caso si verifichino gravi problemi nel computer come, ad esempio, errori in memoria o nei dati che transitano sui bus.
Osservando la Figura 3 e la Figura 6 del precedente capitolo, si puo' notare nella piedinatura della **8086** e della **80386**, la presenza di un terminale chiamato **NMI**; proprio attraverso questo ingresso, le interruzioni non mascherabili vengono inviate alla **CPU**.
Per motivi facilmente comprensibili, si consiglia di tenere **IF** a **0** per il minor tempo possibile; in fase di accensione del **PC**, il flag **IF** viene inizializzato a **1**.

Il flag **DF** o **Direction Flag** (segnalatore di direzione) viene utilizzato in combinazione con le istruzioni della **CPU** che eseguono trasferimenti di dati e comparazioni varie tra due blocchi di memoria; abbiamo gia' visto che in questo caso la **CPU** utilizza **DS:SI** come indirizzo sorgente predefinito, e **ES:DI** come indirizzo destinazione predefinito. Durante queste operazioni, i puntatori **SI** e **DI** vengono automaticamente aggiornati, cioe' incrementati o decrementati; ponendo **DF=0** i puntatori vengono autoincrementati, mentre con **DF=1** i puntatori vengono autodecrementati. In fase di accensione del **PC**, il flag **DF** viene inizializzato a **0**.

10.4 Registri interni delle CPU a 32 bit

Nella famiglia **80x86**, le **CPU** di riferimento per le architetture a **32** bit sono la **80386 DX** e la **80486 DX**; sostanzialmente, la **80486 DX** e' una evoluzione della **80386 DX** con, in piu', un coprocessore matematico incorporato. Nel seguito del capitolo possiamo fare quindi riferimento alla generica sigla **80386**; tutte le considerazioni che verranno esposte, sono perfettamente valide anche per le **CPU** di classe **Pentium** (**80586**, **80686**, etc) a **64** bit.
Una delle caratteristiche fondamentali della **80386**, e' la compatibilita' verso il basso; grazie a questa caratteristica, tutti i programmi scritti per la **8086** possono essere eseguiti, senza nessuna modifica, sulla **80386**. Per ottenere questa compatibilita', la **80386** in fase di accensione del computer, viene inizializzata in modalita' reale; in questa modalita' viene anche disabilitata la linea **A20** dell'**Address Bus**, in modo che la **80386** sia in grado di indirizzare in modo diretto solo 2^{20}**=1048576** byte, proprio come accade per la **8086**.
Naturalmente, tutto cio' non basta per rendere la **80386** pienamente compatibile con la **8086**; e' anche necessario che il set di istruzioni della **80386** sia una estensione del set di istruzioni della **8086**. Analogamente, il set di registri interni della **80386** deve essere una estensione del set di registri interni della **8086**. Tutto cio' che concerne il set di istruzioni delle **CPU 80x86** verra' trattato in dettaglio nel prossimo capitolo; in questo capitolo ci occuperemo, invece, della compatibilita' tra i registri interni della **80386** e della **8086**.
Per ottenere la necessaria compatibilita', si ricorre ad un espediente molto semplice, che consiste nel prendere i registri a **16** bit della **8086**, "estendendoli" a **32** bit nella **80386**; in questo modo si perviene alla situazione illustrata in Figura 7.

Figura 7 - Registri interni della CPU 80386

Osserviamo subito che i registri di segmento della **80386** rimangono a **16** bit; i primi **4** registri, **CS**, **SS**, **DS** e **ES**, sono assolutamente equivalenti agli omonimi registri della **8086**, e in modalita' reale sono destinati a contenere le solite componenti **Seg** dei vari segmenti di programma. Notiamo anche che la **80386** rende disponibili due nuovi registri di segmento, **FS** e **GS**, destinati alla gestione di due ulteriori **Data Segment**; possiamo dire quindi che con la **80386**, il programmatore puo' gestire in modalita' reale ben **4 Data Segment** differenti contemporaneamente. Le sigle **FS** e **GS** di questi due nuovi registri, non hanno nessun significato particolare; si tratta semplicemente della prosecuzione alfabetica delle sigle **DS** e **ES**.

I **5** registri speciali di Figura 7 sono, anche in questo caso, tutti registri puntatori; come si puo' notare, si tratta delle estensioni a **32** bit dei **5** registri puntatori **SI**, **DI**, **BP**, **SP** e **IP** della **8086**. Il programmatore puo' riferirsi ai registri speciali a **32** bit anteponendo una **E** (**extended**) ai nomi dei corrispondenti registri speciali a **16** bit; osserviamo anche che i registri speciali a **16** bit, occupano i **16** bit meno significativi dei corrispondenti registri speciali a **32** bit.

Nota importante.
In un precedente capitolo e' stato detto che con una **CPU** a **32** bit, il programmatore ha la possibilita', in modalita' reale, di servirsi dei registri puntatori a **32** bit per creare coppie logiche del tipo **DS:ESI**, **SS:EBP**, etc; appare evidente pero' che in questo modo, si comprometterebbe la compatibilita' con la **8086** che puo' accettare componenti **Offset** esclusivamente a **16** bit. Proprio per questo motivo, in un programma destinato a girare nella modalita' reale della **80386**, un qualunque offset espresso attraverso un registro puntatore a **32** bit deve essere compreso tra **00000000h** e **0000FFFFh**; in caso contrario, si ottiene un programma che in fase di esecuzione provoca un sicuro crash!

Le considerazioni appena esposte si applicano anche alla gestione dello **Stack Segment** e del **Code Segment**; per indirizzare il **Code Segment** viene utilizzata la coppia **CS:EIP**, ma in modalita' reale, **EIP** contiene un valore che deve essere compreso tra **00000000h** e **0000FFFFh**. In sostanza, ci troviamo nella stessa situazione della **8086** che indirizza il **Code Segment** con la coppia **CS:IP**; di conseguenza, un **Code Segment** di un programma in modalita' reale non puo' superare la dimensione di **65536** byte (**64** Kb).

In relazione infine allo **Stack Segment** bisogna precisare che la **80386** ci permette di gestire uno stack che puo' contenere, sia dati di tipo **WORD**, sia dati di tipo **DWORD**; cio' significa che le istruzioni **PUSH** e **POP** della **80386** possono accettare questa volta, sia operandi di tipo **WORD**, sia operandi di tipo **DWORD**.
Anche in questo caso, e' necessario ribadire che nella modalita' reale della **80386**, l'indirizzamento dello stack deve avvenire con componenti **Offset** comprese tra **00000000h** e **0000FFFFh**; questa condizione deve essere rispettata anche se utilizziamo **ESP** e **EBP**, e cio' significa che lo **Stack Segment** di un programma in modalita' reale non puo' superare la dimensione di **65536** byte (**64** Kb).

Anche nel caso della **80386**, i registri speciali **SI**, **DI**, **ESI** e **EDI** possono svolgere il ruolo, sia di registri puntatori, sia di registri generali; in particolare, utilizzando **ESI** e **EDI** come registri generali, possiamo gestire con grande efficienza generici valori a **32** bit.

I **4** registri generali **EAX**, **EBX**, **ECX** e **EDX** della **80386**, sono le estensioni a **32** bit dei corrispondenti registri a **16** bit **AX**, **BX**, **CX** e **DX** della **8086**; come si puo' notare in Figura 7, vengono supportati, naturalmente, anche gli **half registers** relativi a **AX**, **BX**, **CX** e **DX**. Anche in

questo caso, **BX** e **EBX** possono svolgere il ruolo, sia di registri generali, sia di registri puntatori; come registri puntatori, **BX** e **EBX** vengono normalmente associati a **DS**.

Sulle **CPU 80386** e superiori, le associazioni predefinite tra registri di segmento (**CS**, **DS**, **SS**) e i registri puntatori a **16** bit (**IP**, **SP**, **BP**, **BX**, **SI**, **DI**), seguono le identiche regole gia' esposte per la **8086**.
Le associazioni predefinite tra registri di segmento e i registri puntatori a **32** bit, vengono esposte nel capitolo successivo; per motivi di chiarezza e di compatibilita', si consiglia di evitare gli indirizzamenti a **32** bit nei programmi destinati a girare nella modalita' reale delle **CPU 80386** e superiori.

La **80386** ci permette di utilizzare la sua architettura a **32** bit senza uscire dalla modalita' reale; in questo modo possiamo sfruttare, ad esempio, i registri generali a **32** bit per gestire via hardware operandi a **32** bit da elaborare ad altissima velocita' con la **ALU**. In generale, con la **80386** possiamo gestire via hardware dati a **32** bit che con la **8086** richiederebbero una gestione via software; infatti, con la **8086** un dato a **32** bit deve essere necessariamente scomposto almeno in due parti da **16** bit ciascuna.
Supponiamo con la **80386** di voler copiare il contenuto di **BX** nei **16** bit piu' significativi di **EAX**, e il contenuto di **CX** nei **16** bit meno significativi di **EAX**; un metodo molto sbrigativo per ottenere questo risultato, consiste nell'utilizzare le seguenti tre istruzioni:

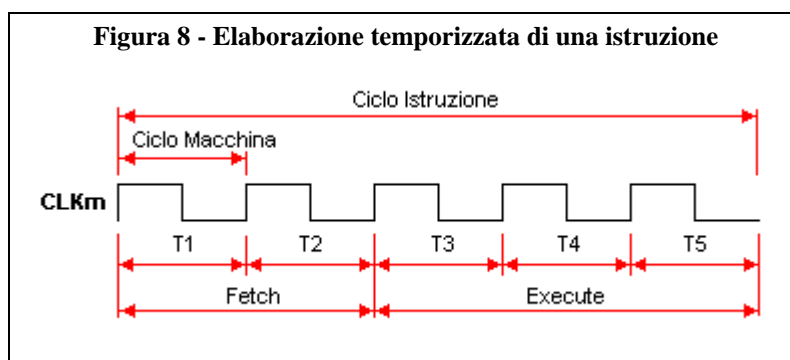
```
PUSH     BX
PUSH     CX
POP      EAX
```

Supponendo che **TOS=0088h**, si vede subito che con le prime due **PUSH**, i **16** bit di **BX** vengono inseriti nello stack agli offset **0086h** e **0087h**, mentre i **16** bit di **CX** vengono inseriti nello stack agli offset **0084h** e **0085h**; a questo punto, l'istruzione **POP** estrae **32** bit dagli offset **0084h**, **0085h**, **0086h**, **0087h** dello stack e li copia in **EAX**. Si noti che dopo l'esecuzione di queste **3** istruzioni, lo stack rimane perfettamente bilanciato; abbiamo effettuato infatti due inserimenti da **16** bit ciascuno e una estrazione da **32** bit.

Passiamo, infine, al **Flags Register** che sulla **80386** viene esteso anch'esso a **32** bit, diventando cosi' l'**Extended Flags Register** o **EFLAGS**; come al solito, per motivi di compatibilita' verso il basso, i primi **16** bit di **EFLAGS** rispecchiano perfettamente la struttura del registro **FLAGS** a **16** bit della **8086**. I successivi **16** bit di **EFLAGS** (e anche i bit inutilizzati in **FLAGS**) contengono altri flags che vengono impiegati in modalita' protetta e verranno descritti nella apposita sezione **Modalita' Protetta** di questo sito; lo stesso discorso vale per numerosi altri registri speciali presenti sulle **80386**, **80486**, etc, che non possono essere utilizzati in modalita' reale.

10.5 Logica di controllo e temporizzazioni della CPU

Abbiamo visto che la **Control Logic** e' il cervello pensante non solo della **CPU**, ma dell'intero computer; istante per istante la **CL** deve stabilire quali dispositivi devono essere disabilitati, quali devono essere invece abilitati e quale compito devono svolgere. Risulta evidente che sarebbe impossibile far funzionare il computer se tutte queste fasi non fossero caratterizzate da una adeguata sincronizzazione e temporizzazione; come abbiamo visto nei precedenti capitoli, questo compito (che consiste in pratica nello scandire il ritmo all'interno del computer) spetta ad un apposito segnale periodico chiamato **clock**.
In base a quanto e' stato esposto nel Capitolo 7, la **CPU** a partire dal segnale di clock di riferimento, ricava altri segnali periodici che servono a gestire diverse sincronizzazioni; in particolare, consideriamo il segnale **CLKm** che assume una notevole importanza. Abbiamo visto infatti che ogni ciclo del segnale **CLKm** rappresenta il cosiddetto **ciclo macchina**, e piu' cicli macchina formano il ciclo istruzione, cioe' il numero di cicli macchina necessari alla **CPU** per eseguire una determinata istruzione; ogni istruzione appartenente ad una **CPU** viene eseguita in un preciso numero di cicli macchina, e questo permette alla logica di controllo di sincronizzare tutte le fasi necessarie per l'elaborazione dell'istruzione stessa.
Le diverse istruzioni che formano un programma, possono differire tra loro in termini di complessita', e questo significa che maggiore e' la complessita' di una istruzione, maggiore sara' il numero di cicli macchina necessari per l'elaborazione; questa situazione viene illustrata dalla Figura 8 che si riferisce al caso di una istruzione che viene elaborata in **5** cicli macchina.

Figura 8 - Elaborazione temporizzata di una istruzione

In linea di principio, l'elaborazione di una istruzione e' costituita da due fasi distinte; la fase di **Fetch** (caricare, prelevare) consiste nella ricerca in memoria e nel successivo caricamento nel **Registro Istruzioni**, del codice macchina della prossima istruzione da eseguire. La fase di **Execute** (esecuzione) consiste nella decodifica dell'istruzione e nella sua elaborazione attraverso la **EU** della **CPU**.
Analizzando in dettaglio l'esempio di Figura 8, si vede che nel ciclo macchina (o fase) **T1** la **CL** compie tutte le azioni necessarie per richiedere alla memoria il codice macchina della prossima istruzione da eseguire; nella fase **T2** viene eseguita la lettura che pone nel **Registro Istruzioni** il codice macchina dell'istruzione stessa. A questo punto la fase di **Fetch** e' terminata; da queste considerazioni si puo' constatare che la fase di **Fetch** richiede sempre due cicli macchina.
La fase di **Execute** invece dipende dalla complessita' dell'istruzione da decodificare ed eseguire; nel caso illustrato in Figura 8, sono necessari tre cicli macchina indicati con **T3**, **T4**, **T5**. Istruzioni molto complesse possono richiedere un numero piuttosto elevato di cicli macchina.

10.6 Prefetch Queue, Cache Memory e Pipeline

Ogni istruzione di un programma, e' rappresentata in memoria da un codice binario formato da un certo numero di bit; decodificando questo codice, la **CPU** acquisisce tutte le informazioni necessarie per eseguire l'istruzione stessa. Il problema che si presenta e' dato dal fatto che per codificare certe istruzioni sono sufficienti pochi bit, mentre per altre istruzioni bisogna ricorrere a codici formati da decine di bit; la soluzione piu' semplice a questo problema, consiste nel rappresentare qualunque istruzione attraverso un codice formato da un numero fisso di bit.
Supponendo ad esempio di utilizzare una codifica a **64** bit, l'esecuzione di un programma da parte della **CPU** viene notevolmente semplificata; in questo caso infatti, la **CPU** legge in sequenza dalla memoria blocchi da **64** bit, ciascuno dei quali contiene il codice macchina di una singola istruzione. Questo codice macchina viene poi decodificato ed eseguito; successivamente la **CPU** passa al prossimo blocco da **64** bit e ripete i passi precedenti (decodifica ed esecuzione).
I computers che utilizzano questa tecnica vengono indicati con la sigla **RISC** o **Reduced Instruction Set Computers** (computers con set semplificato di istruzioni); si tratta di un sistema che semplifica notevolmente il lavoro della **CPU**, ma presenta l'evidente difetto di comportare un notevole spreco di memoria. Osserviamo infatti che per codificare ogni istruzione vengono utilizzati in ogni caso **64** bit; questo accade quindi anche per quelle istruzioni il cui codice macchina richiederebbe invece pochissimi bit.
In contrapposizione ai computers **RISC**, sono nati i computers di tipo **CISC** o **Complex Instruction Set Computers** (computers con set complesso di istruzioni); su questi computers, si utilizza una tecnica piu' sofisticata (e piu' complicata) attraverso la quale il codice macchina specifica non solo il tipo di istruzione, ma anche (in modo implicito) la dimensione in bit del codice stesso. Il vantaggio di questo metodo consiste nel fatto che si ottiene una codifica estremamente compatta, riducendo al minimo lo spreco di memoria; naturalmente, il sistema di decodifica, essendo piu' complesso, porta ad una minore velocita' di elaborazione.
Tutti i **PC** basati sulle **CPU** della famiglia **80x86** sono sostanzialmente di tipo **CISC**; vediamo allora come avviene l'elaborazione delle istruzioni su questo tipo di computers.
Per motivi di efficienza, i codici macchina che rappresentano le varie istruzioni di un programma sono formati da un numero di bit che e' sempre un multiplo di **8**; possiamo dire allora che il codice macchina di una qualsiasi istruzione occupera' in memoria uno o piu' byte. La **CPU** carica il primo byte, e dopo averlo decodificato e' in grado di sapere se quel byte rappresenta l'istruzione completa o se invece e' seguito da altri byte; nel primo caso la **CPU** passa alla fase di elaborazione dell'istruzione, mentre nel secondo caso, la **CPU** provvede a caricare e decodificare gli altri byte che formano l'istruzione. Ad esempio, come vedremo nei capitoli successivi, l'istruzione **Assembly**:

```
ADD AX, Variabile1
```

significa:
`prendi il contenuto della locazione di memoria rappresentata dal nome` **`Variabile1`** `e sommalo al contenuto del registro accumulatore` **`AX`**`, mettendo poi il risultato finale nell'accumulatore stesso.`
L'operando di destra (**Variabile1**) si chiama **operando sorgente**, mentre quello di sinistra (**AX**) si chiama **operando destinazione**.
Supponendo che la componente **Offset** dell'indirizzo in memoria di **Variabile1** sia **0088h**, vedremo in un apposito capitolo che il codice macchina di questa istruzione (in esadecimale) sara':

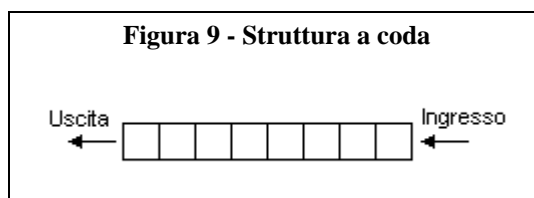
```
03 06 00 88
```

La **CPU** carica il primo byte (**03h**), e dopo averlo decodificato capisce che questo codice significa (come vedremo in seguito):
`esegui la somma tra due operandi a` **`16`** `bit, con l'operando destinazione che e' un registro.`
In base a queste informazioni, la **CPU** sa' che il byte appena elaborato (**03h**) e' seguito sicuramente da un secondo byte (nel nostro caso **06h**) che codifica una serie di informazioni relative al nome del registro, al fatto che l'operando sorgente e' una locazione di memoria e alla modalita' di accesso a

questa locazione; nel nostro caso, il codice **06h** indica che il registro e' **AX** e che l'accesso in memoria avviene direttamente attraverso l'indirizzo (**0088h**) di **Variabile1**.
In base a queste altre informazioni, la **CPU** sa' che dovra' caricare due ulteriori byte (**00h** e **88h**) che rappresentano la componente **Offset** dell'indirizzo di **Variabile1**; a questo punto, la **CPU** ha tutte le informazioni necessarie per elaborare l'istruzione. La **CL** predispone la **ALU** per l'esecuzione di una addizione a **16** bit, dopo aver provveduto a far caricare in un registro temporaneo il contenuto di **AX** e in un altro registro temporaneo il contenuto di **Variabile1** (prelevato dalla memoria all'offset **0088h** del **Data Segment**); la **ALU** esegue la somma e il risultato finale viene messo in **AX**, mentre il registro **FLAGS** conterra' tutte le informazioni sulla operazione appena eseguita.

10.6.1 La Prefetch Queue

Dalle considerazioni appena svolte, si capisce subito che nel caso dei computers **CISC**, la codifica delle istruzioni con un numero variabile di byte crea chiaramente un problema legato al fatto che la **CPU** non puo' conoscere in anticipo la dimensione in bit del codice della prossima istruzione da eseguire; infatti, per avere questa informazione la **CPU** e' costretta a caricare e decodificare il primo byte dell'istruzione, e tutto cio' comporta un notevole aumento degli accessi in memoria con gravi conseguenze sulle prestazioni.
Le vecchie **CPU** come ad esempio l'**8080**, aggiravano il problema caricando il codice macchina delle varie istruzioni, **1** byte alla volta; questo sistema era pero' estremamente inefficiente, e ci si rese subito conto che se si volevano realizzare **CPU** molto piu' veloci, bisognava trovare altre soluzioni.
Con la comparsa sul mercato delle prime CPU a **16** bit (**8086** e **80186**), la **Intel** ha introdotto una prima soluzione a questo problema, ottenendo un notevole incremento delle prestazioni; l'idea e' molto semplice ma geniale allo stesso tempo e consiste nello sfruttare i tempi morti dei bus di sistema (cioe' dell'insieme formato dal **Data Bus**, dal **Control Bus** e dall'**Address Bus**).
Osserviamo infatti che mentre ad esempio la **EU** e' occupata nella fase di esecuzione di una istruzione, puo' capitare che la **BIU** si trovi ad essere inoperosa, in attesa che la **EU** finisca il proprio lavoro; sfruttando questi tempi morti, si puo' impiegare la **BIU** per "pre-caricare" (**prefetch**) dalla memoria una nuova porzione di codice macchina che viene sistemata in un apposito buffer della **CPU**. Non appena la **EU** e' di nuovo libera, la prossima istruzione (o parte di essa) e' gia' pronta per essere elaborata, senza la necessita' di doverla caricare dalla memoria.
Il buffer nel quale si accumulano i vari byte di codice da elaborare ha una capienza adeguata alle esigenze della **CPU**; nel caso ad esempio delle **CPU 8086** e **80186**, la sua dimensione e' di **6** byte.
La struttura piu' adatta per realizzare questo particolare buffer e' sicuramente la cosiddetta **coda** (in inglese **queue**); la Figura 9 illustra le caratteristiche di una generica "coda di attesa".

Figura 9 - Struttura a coda

La coda e' formata da una serie di locazioni destinate a contenere oggetti tutti della stessa natura; si puo' citare come esempio una coda di persone che attendono il loro turno davanti agli sportelli di un Ufficio Postale. La struttura a coda e' dotata di due imboccature che rappresentano l'**Ingresso** e l'**Uscita**; ovviamente, i vari oggetti entrano in ordine dall'ingresso della coda, e si presentano nello stesso ordine all'uscita della coda stessa. Appare anche evidente il fatto che il primo oggetto che esce dalla coda, coincide con il primo oggetto che era entrato nella coda stessa; per questo motivo si dice che la coda e' una struttura di tipo **FIFO** o **First In, First Out** (il primo oggetto ad entrare e' anche il primo ad uscire).

Ogni volta che un oggetto (che sta in testa) esce dalla coda di attesa, libera una locazione permettendo cosi' a tutti gli oggetti che lo precedevano, di avanzare di una posizione; in questo modo si libera anche la locazione che sta in fondo alla coda permettendo cosi' ad un nuovo oggetto di entrare, e cioe' di "accodarsi".

Nel caso delle **CPU** come la **8086**, e' presente come e' stato gia' detto una coda di attesa da **6** byte; ogni volta che la **CPU** estrae dei byte per la decodifica, libera una serie di locazioni che verranno occupate dai byte precedenti. I vari byte in coda avanzano quindi verso la testa, liberando a loro volta diverse posizioni che si trovano in fondo; non appena puo', la **BIU** provvede a caricare dalla memoria altri byte di codice riempiendo i posti rimasti liberi alla fine della coda. Appare chiaro che questo modo di procedere determina un notevole aumento delle prestazioni generali della **CPU**; questo aumento di prestazioni e' legato principalmente al fatto che i vari accessi in memoria, pesantissimi in termini di tempo, vengono effettuati nei tempi morti della **BIU**.
Nel caso delle **CPU**, la struttura di Figura 9 viene chiamata **prefetch queue** (coda di pre-carica).

Per motivi di efficienza, la pre-carica del codice macchina dalla memoria, viene effettuata dalle **CPU** a gruppi di byte; nel caso ad esempio della **8086**, la **BIU** attende che si liberino due posizioni alla fine della coda (**2** byte), dopo di che provvede alla pre-carica dalla memoria di altri due byte (una word) di codice. Tutto cio' significa che nella **prefetch queue** della **8086**, le informazioni da elaborare vengono suddivise in gruppi di **2** byte; di conseguenza, ciascun gruppo risulta allineato ad un indirizzo pari della stessa **prefetch queue**.
La dimensione dei blocchi di codice macchina pre-caricati varia da modello a modello di **CPU**; questa dimensione prende il nome di **allineamento di prefetch** in quanto definisce il tipo di allineamento dei blocchi pre-caricati nella **prefetch queue** (o nella **cache memory**) della **CPU**. La Figura 10 mostra l'allineamento di prefetch nei vari modelli di **CPU**.

Figura 10 - Allineamento di prefetch

CPU	Allineamento
8086	16 bit
80186	16 bit
80286	16 bit
80386 SX	16 bit
80386 DX	32 bit
80486 DX	16 byte (paragrafo)
80586	32 byte

Nel caso ad esempio della **80386 DX**, il codice di un programma viene pre-caricato a blocchi da **32** bit; ogni blocco pre-caricato si trova quindi allineato a indirizzi multipli di **32** bit nella coda di attesa. Nel caso invece della **80486 DX**, il codice di un programma viene pre-caricato a blocchi da **16** byte; ogni blocco pre-caricato si trova quindi allineato a indirizzi multipli di **16** byte nella coda di attesa.
Dal punto di vista del programmatore tutto cio' significa che se si vuole "spremere" al massimo il computer, bisognera' preoccuparsi non solo del corretto allineamento dei dati nei propri programmi, ma anche del corretto allineamento del codice; questi concetti verranno approfonditi nei capitoli successivi.

Un'altro aspetto importante da considerare e' dato dal fatto che la struttura della **prefetch queue** presuppone che le istruzioni che formano un programma vengano eseguite una di seguito all'altra, nello stesso ordine con il quale appaiono disposte in memoria; in effetti questo e' il caso piu'

frequente, ma puo' presentarsi una situazione particolare che determina un "intoppo". Puo' capitare infatti che ad un certo punto della fase di esecuzione di un programma, si arrivi ad una istruzione che prevede un salto ad un'altra zona della memoria relativamente distante dall'istruzione stessa (ad esempio una istruzione che chiama un sottoprogramma); in una situazione del genere il contenuto della **prefetch queue** diventa del tutto inutile in quanto si riferisce ad istruzioni che devono essere "saltate".
In questo caso, l'unica cosa che la **CPU** puo' fare consiste nello svuotare (**flush**) completamente la **prefetch queue** procedendo poi a riempirla con le nuove istruzioni da eseguire; come si puo' facilmente intuire, questa situazione determina un sensibile rallentamento del lavoro che la **CPU** sta eseguendo. Teoricamente, il programmatore puo' eliminare questo problema scrivendo programmi privi di salti; e' chiaro pero' che in pratica sarebbe assurdo procedere in questo modo, anche perche' si andrebbe incontro ad una grave limitazione delle potenzialita' dei programmi stessi.
Senza arrivare a soluzioni cosi' estreme, e' possibile ricorrere a determinati accorgimenti che possono ridurre i disagi per la **CPU**; ad esempio, si puo' allineare opportunamente in memoria l'indirizzo dell'istruzione alla quale si vuole saltare, in modo che la **prefetch queue** possa essere ripristinata il piu' rapidamente possibile.

10.6.2 La Cache Memory

La **prefetch queue** rappresenta a suo modo una piccolissima memoria ad alta velocita' di accesso (**SRAM**) che consente alla **CPU** di ridurre notevolmente i tempi necessari per il caricamento delle varie istruzioni da eseguire; questo aspetto suggerisce l'idea di aumentare adeguatamente le dimensioni della **prefetch queue** in modo che sia possibile caricare in essa una porzione consistente del programma in esecuzione, che verrebbe quindi gestito dalla **CPU** con maggiore efficienza.
Naturalmente pero' bisogna fare i conti con il costo elevato della **SRAM**, e con la sua scarsa densita' di integrazione rispetto alla **DRAM**; i progettisti dei computers, tenendo conto di questi aspetti hanno raggiunto un compromesso rappresentato dalla **cache memory**.
Come abbiamo gia' visto, si tratta di una piccola memoria ad elevate prestazioni, in quanto realizzata con **flip-flop** ad altissima velocita' di risposta; la **cache memory** per le sue ridotte dimensioni non puo' contenere in genere un intero programma, ma solo parte di esso. Ovviamente la domanda che ci poniamo e': quale parte del programma viene caricata nella **cache memory**?
Per rispondere a questa domanda dobbiamo considerare i due aspetti fondamentali che caratterizzano un programma in esecuzione:

1) Nel caso piu' frequente (come abbiamo visto prima) le istruzioni di un programma vengono eseguite nello stesso ordine con il quale appaiono disposte in memoria; si parla a questo proposito di **caratteristica spaziale**.
2) Esistono porzioni di un programma (codice o dati) che statisticamente vengono accedute piu' frequentemente di altre; si parla a questo proposito di **caratteristica temporale**.

La **CPU** sfrutta queste considerazioni, caricando nella **cache memory** le porzioni di codice e dati aventi le caratteristiche appena descritte.
La **cache memory** si trova interposta tra la **CPU** e la memoria centrale; ogni volta che la **CPU** deve caricare una nuova istruzione o nuovi dati del programma, controlla prima di tutto se queste informazioni sono presenti nella **cache memory**. In caso affermativo, la lettura avviene alla massima velocita' possibile; se questa eventualita' si verifica, si parla in gergo di **zero wait states**.
In caso negativo invece, bisognera' necessariamente accedere alla memoria centrale (**DRAM**), che essendo piu' lenta della **cache memory**, comportera' l'inserimento di uno o piu' **wait states**.
Se la **CPU** trova le informazioni desiderate nella cache, si dice in gergo che "la cache e' stata colpita" (**cache hit**), mentre in caso contrario si dice che "la cache e' stata mancata" (**cache miss**); in caso di **cache miss** l'unita' di prefetch della **CPU** provvede a liberare spazio nella cache, eliminando

le informazioni statisticamente meno richieste, riempiendo poi questo spazio con le nuove informazioni lette dalla memoria centrale.
Mentre ad esempio la **8086** dispone di una **prefetch queue** con capienza di soli **6** byte, le attuali **CPU** dispongono di **cache memory** da alcune centinaia di Kb; rispetto alla **prefetch queue**, l'enorme vantaggio della **cache memory** sta nel fatto che in caso di **cache miss**, non e' necessario svuotare tutta la cache, ma solo una parte di essa salvaguardando cosi' tutte le altre informazioni memorizzate. La **80486** ad esempio, in caso di **cache miss** legge dalla memoria centrale un blocco da **16** byte (**1** paragrafo) contenente le nuove informazioni richieste; se nella cache non c'e' spazio, bastera' eliminare solo **16** byte di informazioni non piu' necessarie.
Tutte le **CPU** a partire dalla **80486** dispongono di **cache memory** interna che, come gia' sappiamo, viene chiamata **first level cache** o **L1 cache** (cache di primo livello); la cache interna puo' essere espansa attraverso una cache esterna detta **second level cache** o **L2 cache** (cache di secondo livello).
Anche con la **cache memory** si possono verificare degli intoppi determinati non solo dai salti di un programma, ma anche da altre situazioni che verranno analizzate nei successivi capitoli.

10.6.3 La Pipeline

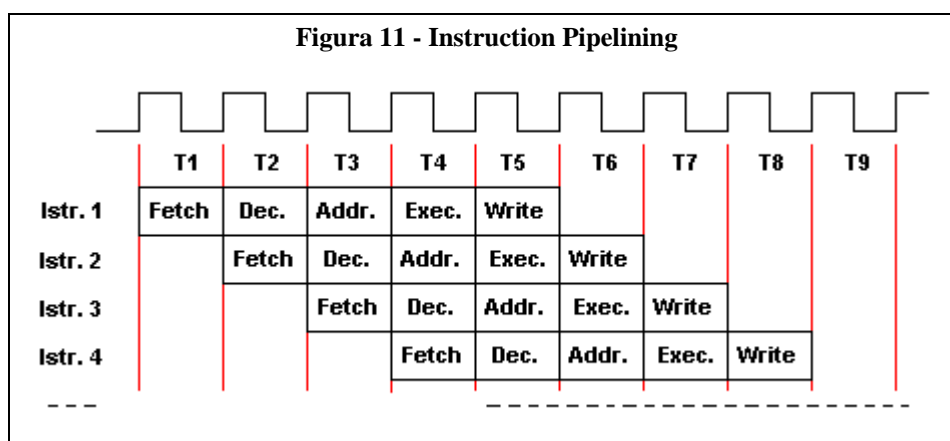
Un'altra importante innovazione che ha determinato un enorme incremento delle prestazioni generali dei microprocessori, e' rappresentata dalla cosiddetta **pipeline**, presente su tutte le nuove **CPU** a partire dalla **80486**; per capire il funzionamento della **pipeline**, si puo' pensare a quello che accade ad esempio nella catena di montaggio di una fabbrica di automobili. Supponiamo per semplicita' che la fabbricazione di una automobile si possa suddividere nelle **5** fasi seguenti:

1) assemblaggio della carrozzeria;
2) verniciatura;
3) allestimento degli interni;
4) installazione del motore;
5) collaudo finale.

Se tutte queste **5** fasi si svolgessero in un unico reparto della fabbrica, si verificherebbe una notevole perdita di tempo causata dalla necessita' di operare su una sola auto per volta; in questo caso, se per realizzare ogni nuova macchina sono necessarie **10** ore di lavoro, possiamo dire che la fabbrica lavora al ritmo di una macchina ogni **10** ore.
Questa situazione puo' essere notevolmente migliorata osservando che le **5** fasi di lavorazione suggeriscono l'idea di predisporre **5** appositi reparti della fabbrica; in questo modo e' possibile operare istante per istante su **5** auto diverse. Non appena la prima auto esce dal reparto assemblaggio per entrare nel reparto verniciatura, il reparto assemblaggio si libera e puo' quindi ospitare la seconda auto; non appena la prima auto esce dal reparto verniciatura per entrare nel reparto allestimento, il reparto verniciatura si libera e puo' ospitare la seconda auto la quale a sua volta libera il reparto assemblaggio che puo' quindi accogliere una terza auto e cosi' via. Prevedendo quindi tanti reparti quante sono le fasi di fabbricazione, si vede subito che la stessa fabbrica puo' lavorare al ritmo di **5** macchine ogni **10** ore, e cioe' **5** volte piu' del caso precedente.

Tutti questi concetti possono essere applicati al microprocessore osservando che (come abbiamo visto in precedenza), anche l'esecuzione di una istruzione puo' essere suddivisa in diverse fasi; nel caso delle vecchie **CPU**, l'esecuzione di una istruzione comportava una lunghissima serie di cicli macchina necessari per la richiesta del codice macchina alla memoria, per il caricamento del codice stesso, per la decodifica, per il caricamento di eventuali operandi dalla memoria e cosi' via.
L'aggiunta di un dispositivo hardware come la **prefetch queue** porta un notevole miglioramento delle prestazioni in quanto consente alla **CPU** di risparmiare tempo; lo scopo della coda di pre-

carica infatti e' quello di fare in modo che la **CPU** trovi la prossima istruzione da eseguire, gia' caricata e pronta per la decodifica (salvo complicazioni legate ad esempio alla precedente esecuzione di una istruzione di salto).
Con l'aggiunta di un altro dispositivo hardware come la **cache memory**, la situazione migliora ulteriormente in quanto la **CPU** trova intere porzioni di un programma gia' caricate e quindi eseguibili alla massima velocita' possibile; grazie alla **cache memory** inoltre, e' spesso possibile evitare intoppi come quelli determinati dalle istruzioni di salto.
Dalle considerazioni appena svolte, nasce l'idea di aggiungere alla **CPU** un numero adeguato di dispositivi hardware, ciascuno dei quali ha il compito di occuparsi di una ben determinata fase della elaborazione di una istruzione; se si organizza allora la **CPU** come una catena di montaggio per automobili, e' possibile ridurre enormemente i tempi di elaborazione grazie al fatto che si puo' operare su piu' istruzioni contemporaneamente, purche' ciascuna di queste istruzioni si trovi in una diversa fase (cioe' in un diverso reparto della catena di montaggio). L'insieme di questi dispositivi formano una cosiddetta **pipeline**; la Figura 11 illustra in modo schematico il principio di funzionamento di una generica **pipeline**.

Figura 11 - Instruction Pipelining

Per chiarire lo schema di Figura 11, suddividiamo innanzi tutto la fase di elaborazione di una istruzione nelle **5** fasi seguenti:

1) Fase di **Fetch**, che consiste nel caricare nel **Registro Istruzioni** della **CPU** il codice macchina della prossima istruzione da eseguire.
2) Fase di **Decodifica**, che consiste nel raccogliere una serie di informazioni riguardanti: il compito che l'istruzione richiede di svolgere alla **CPU**, la presenza di eventuali operandi, etc.
3) Fase di **Addressing** (indirizzamento), che consiste nel predisporre gli indirizzi per caricare gli eventuali operandi.
4) Fase di **Execute** che consiste nell'esecuzione vera e propria dell'istruzione da parte della **CPU**.
5) Fase di **Write Back** che consiste nel salvataggio del risultato nell'operando destinazione (registro o locazione di memoria).

Naturalmente, la **pipeline** viene sincronizzata attraverso un apposito segnale periodico visibile in Figura 11; la stessa Figura 11 mostra la situazione relativa alla esecuzione delle prime **4** istruzioni di un programma.
Nel ciclo **T1** l'istruzione n. **1** si trova nella fase di **Fetch**; nel ciclo **T2** l'istruzione n. **1** passa alla fase di **Decodifica** permettendo alla **pipeline** di procedere con la fase di **Fetch** dell'istruzione n. **2**. Nel ciclo **T3** l'istruzione n. **1** passa all'eventuale fase di **Addressing**, l'istruzione n. **2** passa alla fase di **Decodifica**, mentre l'hardware che si occupa della fase di **Fetch** puo' procedere con l'istruzione n. **3**.
Nel caso illustrato dalla Figura 11 si parla di **pipeline a 5 stadi**; come si puo' notare, in questo caso la **CPU** puo' operare istante per istante su piu' istruzioni differenti. Piu' aumentano gli stadi della

pipeline, maggiore sara' il numero di istruzioni elaborate in contemporanea e minore sara' il tempo richiesto per ogni fase; in piu' bisogna anche considerare un ulteriore aumento delle prestazioni dovuto al fatto che spesso alcune delle fasi descritte prima non sono necessarie (caricamento degli operandi, write back).

Anche con la **pipeline** possono verificarsi degli imprevisti che determinano un certo rallentamento delle operazioni. Supponiamo che nella Figura 11, l'istruzione n. **1** preveda nella fase di **Write Back** la scrittura in memoria del risultato finale, mentre l'istruzione n. **3** preveda nella fase di **Addressing** il caricamento di un operando da una locazione di memoria; come si puo' notare, si viene a creare un problema in quanto entrambe le operazioni dovrebbero svolgersi nel ciclo **T5**. La **CPU** non puo' effettuare contemporaneamente due accessi in memoria (uno in lettura e uno in scrittura), e deve quindi "accontentare" una sola delle due istruzioni; in un caso del genere, la **pipeline** non puo' fare altro che eseguire nel ciclo **T5** la fase di **Write Back** per l'istruzione n. **1**, rimandando la fase di **Addressing** per l'istruzione n. **3** al ciclo **T6**. La fase di **Addressing** per l'istruzione n. **3** richiede quindi non **1** ma **2** cicli del segnale di Figura 11; questa situazione rappresenta una cosiddetta fase di "stallo" della **pipeline** (**pipeline stall**).
Le **CPU** di classe **80586** e superiori, utilizzano due o piu' **pipeline** attraverso le quali possono eseguire in alcuni casi piu' istruzioni in contemporanea; questa situazione si verifica ad esempio quando la **CPU** incontra due istruzioni consecutive, indipendenti l'una dall'altra e tali da non provocare stalli nella **pipeline**. Se si verificano queste condizioni, e' possibile eseguire le due istruzioni in parallelo; le **CPU** capaci di eseguire piu' istruzioni in parallelo vengono definite **superscalari**.
Una potentissima caratteristica delle **CPU** superscalari e' data dalla possibilita' di poter "indovinare" la direzione piu' probabile, lungo la quale proseguira' l'esecuzione di un programma; in sostanza, in presenza di una istruzione di salto "condizionato", la **CPU** puo' tentare di indovinare se il salto verra' effettuato o meno. Questa caratteristica viene definita **branch prediction** e consente in moltissimi casi di evitare lo svuotamento della pipeline in presenza di una istruzione di salto.
In pratica, per alimentare ciascuna pipeline vengono utilizzate due code di prefetch; la prima viene riempita nel solito modo (leggendo le istruzioni dalla memoria, una di seguito all'altra), mentre la seconda viene invece riempita in base alle "previsioni" fatte da un apposito algoritmo di branch prediction. Se nella prima coda viene incontrata una istruzione di salto e se l'algoritmo di branch prediction ha azzeccato le previsioni, allora la seconda coda conterra' le nuove istruzioni che ci servono; in questo modo, si evita lo svuotamento delle pipeline e l'esecuzione non subisce nessun rallentamento.

I programmatori che volessero approfondire questi concetti, possono consultare la documentazione tecnica messa a disposizione dai vari produttori di **CPU** (vedere a tale proposito la sezione **Links Utili**).

Capitolo 11 - Il codice macchina e il codice Assembly

Fortunatamente, un programmatore **Assembly** non deve scrivere i suoi programmi utilizzando il codice macchina (sarebbe piuttosto complicato); nonostante cio', i concetti che vengono esposti in questo capitolo si possono considerare di vitale importanza per chi ambisce ad acquisire una completa padronanza nell'utilizzo di questo potentissimo strumento di programmazione.

Nei precedenti capitoli e' stato detto che le prime macchine di calcolo automatico, venivano progettate incorporando al loro interno l'unico programma che potevano eseguire; se si aveva la necessita' di eseguire un programma differente, bisognava procedere alla progettazione di una nuova macchina. Per ovviare a questa situazione piuttosto scomoda, si penso' di realizzare delle macchine che incorporavano al loro interno un certo numero di programmi, selezionabili attraverso meccanismi piu' o meno complicati; in questo modo si migliorava sicuramente la situazione, ma non si risolveva certo il problema di fondo rappresentato dalla scarsissima flessibilita' di questo sistema. In particolare, si rendeva necessario limitare al massimo il numero di programmi che queste macchine potevano eseguire, per evitare di andare incontro ad enormi complicazioni costruttive; in questo tipo di macchine si puo' dire che la logica di calcolo viene "cablata" all'interno della macchina stessa, e per questo motivo si parla anche di **logica cablata**.

Negli attuali computers, si utilizza invece un altro metodo che permette di eliminare radicalmente i problemi appena citati; invece di adattare il computer al programma da eseguire, si fa' esattamente il contrario, adattando cioe' il programma da eseguire al computer. Questo obiettivo viene raggiunto grazie al fatto che il cuore del computer, e cioe' la **CPU**, presenta come sappiamo la caratteristica di poter riconoscere ed interpretare un certo numero di istruzioni a ciascuna delle quali corrisponde una ben precisa azione che la **CPU** stessa puo' svolgere; il programmatore dialoga con la **CPU** attraverso un programma che puo' essere visto come un insieme di dati e istruzioni. L'elaborazione di un programma da parte del computer consiste nell'inviare in sequenza queste istruzioni alla **CPU**, la quale puo' cosi' procedere alla fase di decodifica e di esecuzione; grazie a questa tecnica, utilizzando poche decine di istruzioni e' possibile scrivere una quantita' di programmi limitata solo dalla fantasia del programmatore. Attraverso questo sistema la **CPU** puo' essere programmata e riprogrammata a piacere, e per questo motivo si parla anche di **logica programmabile**.

Ogni famiglia di **CPU** quindi e' in grado di decodificare ed eseguire un ben determinato insieme (**set**) di istruzioni; la totalita' di queste istruzioni forma il cosiddetto **set di istruzioni della CPU**. Nei precedenti capitoli abbiamo visto che la compatibilita' tra diverse famiglie di **CPU** viene ottenuta rendendo compatibili le loro architetture interne; chiaramente pero' questo non basta, in quanto e' necessario che le diverse famiglie di **CPU** siano compatibili anche a livello di set di istruzioni. Da questo punto di vista, due **CPU** sono compatibili tra loro quando hanno lo stesso set di istruzioni, oppure quando il set di istruzioni di una **CPU** e' un sottoinsieme del set di istruzioni dell'altra **CPU**; questa e' proprio la strada seguita per garantire la compatibilita' verso il basso tra i diversi modelli delle **CPU 80x86**. Possiamo dire quindi che l'**80286** estende il set di istruzioni dell'**8086**, l'**80386** estende il set di istruzioni dell'**80268**, l'**80486** estende il set di istruzioni dell'**80386** e cosi' via.

Come gia' sappiamo, l'unico linguaggio che la **CPU** e' in grado di capire e' il linguaggio binario; cio' implica che tutte le informazioni (codice, dati e stack) presenti in un programma, prima di poter essere elaborate, devono essere convertite in codice binario. Questo lavoro viene svolto da appositi strumenti software chiamati **traduttori**; una volta che il traduttore ha svolto il suo compito, il nostro programma si presenta sotto forma di sequenza di numeri binari che nel loro insieme formano il cosiddetto **machine code** (codice macchina).

I programmi traduttori si suddividono in: **assemblatori**, **compilatori** e **interpreti**; l'assemblatore o assembler, ha il compito di tradurre in codice macchina un programma scritto in **Assembly**. Il compilatore o compiler, ha il compito di tradurre in codice macchina un programma scritto con un linguaggio di alto livello che per questo motivo viene chiamato **linguaggio compilato**; come vedremo in seguito, esiste una differenza abissale tra il lavoro svolto da un assemblatore e il lavoro svolto da un compilatore. L'interprete o interpreter, e' analogo al compilatore, con la differenza che le istruzioni del programma vengono convertite in codice macchina ed eseguite una alla volta; i linguaggi di alto livello che permettono di scrivere programmi destinati agli interpreti, si chiamano **linguaggi interpretati**.
Mentre gli assemblatori e i compilatori generano un programma direttamente eseguibile dalla **CPU**, gli interpreti generano programmi che per essere eseguiti richiedono la presenza dell'interprete stesso in memoria; questo accade ad esempio con i programmi scritti nel vecchio **BASIC interpretato** che possono essere eseguiti solo se si ha a disposizione un apposito interprete **BASIC**.
Sia i compilatori che gli interpreti presentano vantaggi e svantaggi che verranno analizzati in seguito.

Il problema che si pone quindi, consiste nel trovare il modo per convertire in codice binario le informazioni (codice, dati e stack) che formano un programma; vediamo allora quale metodo viene seguito per raggiungere questo obiettivo.

11.1 Caratteristiche generali delle istruzioni per le CPU 80x86

Una qualunque istruzione, per poter essere eseguita, deve fornire alla **CPU** una serie di informazioni fondamentali che comprendono in particolare il tipo di operazione da svolgere; in sostanza, attraverso una istruzione dobbiamo dire alla **CPU** se vogliamo che venga eseguita una addizione, una divisione, una moltiplicazione, un complemento a **1**, un trasferimento dati, etc.
Come si puo' facilmente intuire, molte di queste operazioni che la **CPU** deve eseguire, richiedono la presenza implicita o esplicita di appositi dati che le operazioni stesse devono elaborare; l'addizione ad esempio richiede la presenza di due addendi, cosi' come il trasferimento dati richiede una sorgente e una destinazione per i dati stessi.
Le entita' come: gli addendi di una addizione, i fattori di una moltiplicazione, la sorgente e la destinazione di un trasferimento dati, etc, vengono chiamati, come gia' sappiamo, **operandi**; un qualsiasi operando puo' essere rappresentato da:

*** un registro**
*** una locazione di memoria**
*** una costante numerica**

Nel seguito del capitolo e nei capitoli successivi, per indicare il tipo di operandi coinvolti da una istruzione faremo uso per comodita' di apposite abbreviazioni.

Un generico operando di tipo registro generale o registro speciale come **AX**, **BL**, **SI**, **BP**, etc, viene indicato con l'abbreviazione **Reg**; nel caso in cui sia necessario specificare la dimensione in bit del registro, si usano le abbreviazioni **Reg8**, **Reg16**, **Reg32**, etc. Come gia' sappiamo, il programmatore puo' accedere direttamente in lettura o in scrittura ai registri interni della **CPU**; tutto cio' significa in sostanza che il contenuto di un registro interno della **CPU** puo' essere modificato in qualsiasi momento attraverso le istruzioni di un programma.

Un generico operando di tipo registro di segmento come **CS**, **SS**, etc, viene indicato con l'abbreviazione **SegReg**; come gia' sappiamo, i registri di segmento delle **CPU 80x86** sono tutti a **16** bit, per cui il simbolo **SegReg** equivale a **SegReg16**. Anche nel caso dei registri di segmento della

CPU, il programmatore puo' accedere ad essi in lettura o in scrittura; in questo caso pero' la situazione e' chiaramente piu' delicata, e verra' analizzata in dettaglio nel seguito del capitolo e nei capitoli successivi.

Un generico operando rappresentato da una locazione di memoria come **DS:[BX]**, **SS:[BP]**, etc, viene indicato con l'abbreviazione **Mem**; nel caso in cui sia necessario specificare la dimensione in bit della locazione di memoria, si usano le abbreviazioni **Mem8**, **Mem16**, **Mem32**, etc. Un operando di tipo **Mem** e' associato ad una locazione di memoria individuata da un ben preciso indirizzo **Seg:Offset** (come **DS:BX**); il programmatore puo' quindi accedere in lettura o in scrittura a questo indirizzo modificandone il contenuto.

Un generico operando rappresentato da una costante numerica come **-121**, **12d**, **3F2Ch**, etc, cioe' da un valore numerico immediato, viene indicato con l'abbreviazione **Imm**; nel caso in cui sia necessario specificare la dimensione in bit del valore immediato, si usano le abbreviazioni **Imm8**, **Imm16**, **Imm32**, etc. Un operando di tipo **Imm**, essendo un semplice valore numerico, non e' associato a nessuna locazione di memoria, e quindi non puo' essere modificato dal programmatore; come verra' spiegato in dettaglio nel seguito del capitolo, i valori numerici immediati vengono "incorporati" direttamente nel codice macchina dei programmi.

11.1.1 Combinazioni permesse tra gli operandi di una istruzione

Nel caso piu' generale possibile, una istruzione destinata ad una **CPU 80x86** richiede la presenza di due operandi che vengono sempre chiamati **sorgente** e **destinazione**; questa terminologia viene quindi usata anche per indicare i due addendi di una addizione, i due fattori di una moltiplicazione, il dividendo e il divisore di una divisione, etc.
Alcune istruzioni vengono elaborate dalla **CPU** attraverso l'uso di operandi impliciti che non devono essere quindi specificati dal programmatore; nella gran parte dei casi pero', il programmatore ha il compito di specificare in modo esplicito l'operando o gli operandi da impiegare con una istruzione.

A seconda del tipo di istruzione da elaborare, le **CPU** della famiglia **80x86** permettono solamente determinate combinazioni tra operando sorgente e operando destinazione; nel caso ad esempio dell'addizione, la **CPU** somma il contenuto dell'operando sorgente con il contenuto dell'operando destinazione, e mette il risultato nello stesso operando destinazione (il cui vecchio contenuto viene quindi sovrascritto). Per l'addizione le **CPU 80x86** permettono solamente le seguenti combinazioni tra sorgente e destinazione:

1) Somma tra sorgente **Reg** e destinazione **Reg**
2) Somma tra sorgente **Mem** e destinazione **Reg**
3) Somma tra sorgente **Reg** e destinazione **Mem**
4) Somma tra sorgente **Imm** e destinazione **Reg**
5) Somma tra sorgente **Imm** e destinazione **Mem**

Osserviamo subito che l'addizione non puo' coinvolgere come operandi i registri di segmento; osserviamo anche che non e' consentita la somma tra **Mem** e **Mem**. Questa e' una regola generale legata al fatto che le **CPU 80x86** richiedono che tutte le operazioni si svolgano sotto la supervisione della **CPU** stessa; l'unica eccezione e' rappresentata dalla operazione di trasferimento dati che puo' essere effettuata da **Mem** a **Mem** attraverso la tecnica del **DMA** o **Direct Memory Access**.

Un altro aspetto abbastanza intuitivo, e' legato al fatto che in generale, sono proibite tutte le combinazioni prive di senso, come ad esempio l'uso di un operando di tipo **Imm** come destinazione;

infatti, come e' stato detto in precedenza, un operando di tipo **Imm** non e', ne' un registro, ne' una locazione di memoria, per cui la **CPU** non puo' accedere ad esso per modificarne il contenuto.

Un'altra importantissima regola generale, riguarda il fatto che gli operandi di una istruzione devono essere "compatibili" tra loro in termini di ampiezza in bit; nel caso della addizione, non avrebbe nessun senso pensare di sommare ad esempio il contenuto a **16** bit di **AX** con il contenuto a **8** bit di **DL**. Una **CPU** a **16** bit puo' sommare tra loro due operandi che devono essere, o entrambi a **8** bit, o entrambi a **16** bit; analogamente, una **CPU** a **32** bit puo' sommare tra loro due operandi che devono essere, o entrambi a **8** bit, o entrambi a **16** bit, o entrambi a **32** bit.

Per quanto riguarda invece il trasferimento dati, questa operazione consiste nel trasferire il contenuto dell'operando sorgente nel contenuto dell'operando destinazione; per questa istruzione (che e' quella usata in modo piu' massiccio nei programmi **Assembly**), le **CPU 80x86** permettono solamente le seguenti combinazioni tra sorgente e destinazione:

1) Trasferimento dati da **Reg** a **Reg**
2) Trasferimento dati da **SegReg** a **Reg**
3) Trasferimento dati da **Reg** a **SegReg**
4) Trasferimento dati da **Mem** a **Reg**
5) Trasferimento dati da **Reg** a **Mem**
6) Trasferimento dati da **Mem** a **SegReg**
7) Trasferimento dati da **SegReg** a **Mem**
8) Trasferimento dati da **Imm** a **Reg**
9) Trasferimento dati da **Imm** a **Mem**

Come e' stato gia' detto, se vogliamo effettuare un trasferimento da **Mem** a **Mem** che coinvolge un blocco dati di grosse dimensioni, possiamo ricorrere alla tecnica del **DMA**; questo argomento verra' trattato nella sezione **Assembly Avanzato**.

Nei precedenti capitoli abbiamo gia' incontrato le due istruzioni **PUSH** e **POP** che fanno chiaramente parte della categoria delle istruzioni per il trasferimento dati; abbiamo anche visto che per ciascuna di queste due istruzioni il programmatore deve indicare un solo operando esplicito, in quanto l'altro operando e' rappresentato implicitamente da una locazione di memoria che si trova nello stack.
L'istruzione **PUSH** richiede esplicitamente il solo operando sorgente; l'operando destinazione infatti e' implicitamente rappresentato da una locazione di memoria dello stack puntata da **SS:SP**.
L'istruzione **POP** richiede esplicitamente il solo operando destinazione; l'operando sorgente infatti e' implicitamente rappresentato da una locazione di memoria dello stack puntata da **SS:SP**.

Nel caso del trasferimento dati si nota in modo ancora piu' evidente che non avrebbe nessun senso utilizzare un operando di tipo **Imm** come destinazione; analogamente, non avrebbe senso pensare di trasferire ad esempio il contenuto a **16** bit di **CX** negli **8** bit di **AL**.
Nei successivi capitoli verranno analizzate in dettaglio tutte le istruzioni delle **CPU 80x86** con le relative combinazioni valide tra operando sorgente e operando destinazione.

11.1.2 Operando di riferimento

Abbiamo appena visto che un operando di una istruzione puo' essere di tipo **Reg**, **SegReg**, **Mem** o **Imm**; tralasciando per il momento gli operandi di tipo **SegReg** che rappresentano casi particolari, bisogna dire che le **CPU 80x86**, nelle istruzioni, privilegiano un eventuale operando di tipo **Reg** trattandolo come **operando di riferimento**. Osserviamo subito che ad eccezione delle istruzioni con sorgente **Imm** e destinazione **Mem**, in tutti gli altri casi e' sicuramente presente almeno un operando di tipo **Reg**; si possono presentare allora due possibilita':

1) Entrambi gli operandi sono di tipo **Reg**; in questo caso, per convenzione l'operando di riferimento e' il registro destinazione.
2) Uno solo degli operandi e' di tipo **Reg**; in questo caso, l'operando di riferimento e' ovviamente quello di tipo **Reg** che puo' trovarsi, o alla sorgente, o alla destinazione.

Tutte le istruzioni per le quali l'operando di riferimento e' un registro destinazione, vengono definite **to register** (operazioni **verso registro** destinazione); tutte le istruzioni per le quali l'operando di riferimento e' un registro sorgente, vengono definite **from register** (operazioni **da registro** sorgente). Ad esempio, un trasferimento dati da **Mem** a **Reg** e' una operazione **to register**; l'operando di riferimento infatti e' il registro destinazione. Analogamente, una addizione tra sorgente **Reg** e destinazione **Mem** e' una operazione **from register**; l'operando di riferimento infatti e' il registro sorgente.

11.2 Codice macchina delle istruzioni 8086

Come al solito, partiamo dal caso della **8086** che e' la **CPU** di riferimento per le architetture a **16** bit della famiglia **80x86**; vediamo quindi come viene effettuata dall'assembler la codifica binaria delle istruzioni destinate a questa **CPU**.

Abbiamo gia' visto che i computers equipaggiati con le **CPU 80x86** fanno parte della categoria denominata **CISC** o **Complex Instruction Set Computers**; su questo tipo di **CPU** viene utilizzata una sofisticata tecnica di codifica delle istruzioni, che permette di ottenere un codice macchina estremamente compatto. Ogni istruzione viene convertita in un codice che puo' occupare uno o piu' byte; la **CPU** legge il primo byte, e dopo averlo decodificato e' in grado di sapere come procedere per acquisire tutte le informazioni necessarie per l'elaborazione dell'istruzione stessa.
In questo capitolo verranno illustrati gli aspetti generali, relativi alla codifica delle istruzioni; chi volesse approfondire l'argomento, puo' consultare la documentazione tecnica messa a disposizione dalla **Intel** nel suo sito **WEB** (vedere a tale proposito la sezione **Links Utili**). In particolare, si consiglia di scaricare il documento **231455.PDF** che reca il titolo **8086 16 BIT HMOS MICROPROCESSOR**, e il documento **24319101.PDF** che reca il titolo **Intel Architecture Software Developer's Manual - Volume 2 - Instruction Set Reference**; la quasi totalita' della documentazione e' disponibile in formato **Acrobat PDF** (e in lingua inglese).

11.2.1 Struttura generale di una istruzione in codice macchina

La prima cosa da dire riguarda il fatto che ogni istruzione di un programma contiene una serie di elementi (o **campi**) che al momento dell'esecuzione dovranno essere riconosciuti dalla **CPU**; lo scopo di questi campi e' quello di codificare svariate informazioni che comprendono tra l'altro: il tipo (categoria) di istruzione da eseguire, gli eventuali registri coinvolti, gli indirizzi di memoria degli eventuali dati coinvolti, la modalita' di accesso in memoria ai dati stessi e cosi' via. Per fare in modo che la **CPU** sia in grado di riconoscere questi campi, dobbiamo assegnare a ciascuno di essi un apposito codice binario; combinando opportunamente tra loro questi codici binari, si ottiene il

codice macchina dell'istruzione da eseguire.
Prima di tutto bisogna quindi definire la struttura generale che assume una istruzione appartenente al set di istruzioni della **CPU 8086**; come e' stato gia' detto, ogni istruzione e' formata da una serie di campi, ciascuno dei quali deve trovarsi in una posizione ben determinata. La **CPU** elabora in sequenza questi campi ricavando da ciascuno di essi le informazioni necessarie per poter procedere con la fase di elaborazione; attraverso la Figura 1 possiamo analizzare la struttura generale che assume una istruzione, valida per le **CPU 8086**.

Figura 1 - Struttura di una istruzione 8086

Campo	Dimensione in byte
Instruction Prefix	0 o 1
Segment Override Prefix	0 o 1
Opcode	1
mod_reg_r/m	0 o 1
Displacement	0, 1 o 2
Immediate	0, 1 o 2

I vari campi di Figura 1 sono ordinati dall'alto verso il basso, e vengono disposti in sequenza in memoria in modo da formare il codice macchina di una istruzione; analizziamo in dettaglio il significato di ciascun campo.

11.2.2 Campo Prefisso (Instruction Prefix e Segment Override Prefix)

Come si puo' notare, i primi **2** campi sono dei prefissi opzionali; ciascun prefisso, se e' presente, occupa **1** byte. Lo scopo dei prefissi e' quello di fornire alla **CPU** una serie di informazioni relative alle caratteristiche dei successivi campi che formano l'istruzione; come vedremo nel seguito del capitolo, i prefissi svolgono un ruolo fondamentale per ottenere la compatibilita' verso il basso tra le **CPU** della famiglia **80x86**. La Figura 2 illustra l'elenco completo dei codici macchina (in binario e in esadecimale) di tutti i prefissi disponibili; questi prefissi possono essere disposti in un ordine qualsiasi, ma devono trovarsi rigorosamente all'inizio del codice macchina di una istruzione.

Figura 2 - Codici macchina dei prefissi

Prefisso		Codice macchina	
		HEX	BIN
Instruction Prefix	LOCK	F0h	11110000b
	REPNE/REPNZ	F2h	11110010b
	REP, REPE/REPZ	F3h	11110011b
Segment Override Prefix	ES	26h	00100110b
	CS	2Eh	00101110b
	SS	36h	00110110b
	DS	3Eh	00111110b

L'**Instruction Prefix**, se e' presente, indica alla **CPU** che l'istruzione da elaborare e' preceduta da uno tra i possibili prefissi **LOCK**, **REPNE/REPNZ**, **REP**, **REPE/REPZ**; questi prefissi vengono usati insieme a particolari istruzioni che saranno analizzate in dettaglio in un capitolo successivo.

Il **Segment Override Prefix**, se e' presente, indica alla **CPU** che l'istruzione da elaborare contiene un indirizzo di memoria la cui componente **Offset** deve essere riferita ad un registro di segmento specificato dal prefisso stesso; come gia' sappiamo, il programmatore deve ricorrere al **segment override** per aggirare le associazioni predefinite che la **CPU** effettua tra registri puntatori e registri di segmento. Ogni volta che in una istruzione e' presente un segment override, l'assembler inserisce uno dei codici da **1** byte visibili in Figura 2; cio' significa che ogni segment override fa' crescere di **1** byte le dimensioni del codice macchina del programma!

11.2.3 Campo Opcode

Subito dopo gli eventuali prefissi, troviamo un campo che occupa **1** byte ed e' quindi sempre presente nel codice macchina di una istruzione; questo campo viene chiamato **Opcode** e assume una importanza enorme. Infatti, il campo **Opcode** contiene al suo interno un valore binario che codifica il tipo di operazione che la **CPU** deve eseguire; attraverso l'**Opcode** quindi, possiamo indicare alla **CPU** se vogliamo che venga eseguita una addizione, una sottrazione, un prodotto, un trasferimento di dati, una chiamata di un sottoprogramma, etc. La **Control Logic**, in base al codice contenuto nell'**Opcode**, predispone la **ALU** o gli altri dispositivi della **CPU** affinche' eseguano il compito richiesto. La Figura 3 illustra la struttura interna piu' frequente del campo **Opcode**; come vedremo nel seguito del capitolo, in certi casi questo campo puo' assumere anche altre configurazioni.

Figura 3 - Campo Opcode

Contenuto	x	x	x	x	x	x	d	w
Posizione bit	7	6	5	4	3	2	1	0

Nel caso piu' semplice, una istruzione puo' essere interamente codificata con i soli **8** bit del campo **Opcode**; il codice **10011100** ad esempio, e' una istruzione completa che indica alla **CPU** di salvare nello stack il contenuto del registro dei flags (**FLAGS**). In un caso del genere, la **CPU** non ha bisogno di ulteriori informazioni in quanto sa' gia' come si deve comportare; in sostanza, quando la **CPU** incontra l'**Opcode 10011100**, legge il contenuto a **16** bit del registro **FLAGS** e lo copia nella locazione di memoria a **16** bit che si trova in cima allo stack ad un indirizzo logico che viene gestito, come sappiamo, attraverso la coppia **SS:SP**. Possiamo dire allora che in questa istruzione, il registro **FLAGS** ricopre il ruolo di **operando sorgente** "implicito", mentre la locazione di memoria puntata da **SS:SP** ricopre il ruolo di **operando destinazione** "implicito"; una osservazione molto importante riguarda il fatto che il trasferimento dati appena illustrato si svolge da una sorgente a **16** bit a una destinazione che, come e' stato gia' detto, deve essere ugualmente a **16** bit.

Abbiamo visto pero' che nel caso generale, la situazione e' piu' complessa in quanto la quasi totalita' delle istruzioni della **CPU** richiede uno o due operandi che devono essere esplicitamente indicati dal programmatore; per gestire questa situazione, l'**Opcode** di Figura 3 viene suddiviso in due parti principali.
Nel caso piu' frequente, i **6** bit piu' a sinistra (posizioni dalla **2** alla **7**) indicati con **x**, contengono il codice binario vero e proprio del comando da inviare alla **CPU**; ad esempio, la sequenza **100010b** codifica una richiesta di trasferimento dati, la sequenza **000000b** codifica l'operazione di somma, la sequenza **111101b** codifica l'operazione di complemento a **1** (**NOT**).
Attraverso questi tre esempi si puo' osservare appunto che molto spesso le istruzioni prevedono la presenza di operandi espliciti; il trasferimento dati ad esempio avviene da una sorgente ad una destinazione. L'addizione prevede la presenza di due addendi; il complemento a **1** si applica ovviamente ad un numero binario.
In base a queste osservazioni, risulta evidente che nel caso in cui sia prevista la presenza di operandi

espliciti, bisognera' fornire alla **CPU** ulteriori informazioni; queste informazioni devono permettere alla **CPU** di poter ricavare una serie di dettagli relativi ad esempio al tipo di operandi (**Reg**, **Mem** o **Imm**), alla loro dimensione in bit, etc.

A proposito della dimensione in bit degli eventuali operandi di una istruzione, bisogna ribadire ancora una volta che non avrebbe senso ad esempio sommare un operando a **8** bit con un operando a **16** bit; analogamente, non avrebbe senso trasferire il contenuto del registro **AX** (**16** bit) nel registro **BL** (**8** bit).
Una **CPU 8086** con architettura a **16** bit puo' gestire via hardware operandi a **8** bit o a **16** bit; per indicare alla **CPU** la dimensione in bit degli operandi, si utilizza il bit **w** o **word bit** che nell'**Opcode** di Figura 3 si trova in posizione **0**. Se **w=0**, gli operandi sono a **8** bit; se invece **w=1**, gli operandi sono a **16** bit.

Il bit che nell'**Opcode** di Figura 3 occupa la posizione **1** ed e' indicato con la lettera **d**, rappresenta il cosiddetto **direction bit**; questo bit viene utilizzato per indicare l'operando di riferimento di cui si e' gia' parlato nella sezione **11.1**.
Abbiamo gia' visto che se una istruzione prevede la presenza di due operandi, entrambi di tipo **Reg**, allora l'operando di riferimento e' per convenzione il registro destinazione; se una istruzione prevede la presenza di due operandi, e uno solo di essi e' di tipo **Reg**, allora quel registro viene trattato ovviamente come operando di riferimento. Per indicare alla **CPU** quale ruolo (sorgente o destinazione) svolge l'operando registro di riferimento, viene appunto utilizzato il bit **d**; se **d=0**, l'operando (registro) di riferimento svolge il ruolo di operando sorgente (operazione **from register**), mentre se **d=1**, l'operando (registro) di riferimento svolge il ruolo di operando destinazione (operazione **to register**).
Tutti questi aspetti verranno chiariti meglio attraverso gli esempi pratici presentati piu' avanti; si tenga anche presente che, come vedremo nel seguito del capitolo, i due bit **d** e **w** possono assumere in alcune circostanze un significato differente da quello appena descritto.

11.2.4 Campo mod_reg_r/m

Abbiamo visto che attraverso il campo **Opcode**, la **CPU** ricava una serie di informazioni generali sugli operandi di una istruzione; l'**Opcode 100010dw** ad esempio, indica un trasferimento dati che coinvolge operandi di tipo **Reg** o **Mem**. L'**Opcode 1100011w** indica un trasferimento dati da sorgente **Imm** a destinazione **Reg** o **Mem**; l'**Opcode 10001110** indica un trasferimento dati da sorgente **Reg** o **Mem** a destinazione **SegReg**.
Nel caso di istruzioni che prevedono la presenza di operandi esplicitamente specificati dal programmatore, il codice da **1** byte che rappresenta l'**Opcode** non e' sufficiente per fornire alla **CPU** tutte le informazioni necessarie per l'elaborazione dell'istruzione stessa; la **CPU** infatti deve anche conoscere il nome di eventuali operandi di tipo **Reg**, la modalita' di accesso ad eventuali operandi di tipo **Mem**, etc.
In una situazione di questo genere, l'**Opcode** e' seguito da un secondo codice binario da **1** byte che ha proprio lo scopo di fornire alla **CPU** tutte le informazioni supplementari; in Figura 1 questo ulteriore campo viene indicato con **mod_reg_r/m**.
Riprendendo un esempio precedente, abbiamo visto che l'**Opcode** per l'addizione tra operandi **Reg** o **Mem** vale **000000dw**; ponendo **d=1** e **w=1** otteniamo l'**Opcode 00000011** che indica alla **CPU** che bisogna eseguire una addizione tra due operandi a **16** bit (**w=1**), con un registro che ricopre il ruolo di operando destinazione e verra' quindi trattato come operando di riferimento (**d=1**). In un caso del genere, la **CPU** ha bisogno di sapere qual'e' il registro da usare come destinazione, e se l'operando sorgente e' un altro registro oppure una locazione di memoria. Inoltre, se l'operando sorgente e' un altro registro, bisogna dire alla **CPU** di quale registro si tratta; se invece l'operando sorgente e' una locazione di memoria, bisogna dire alla **CPU** come avviene l'accesso a questa

locazione.

Come e' stato gia' detto, tutte queste informazioni vengono passate alla **CPU** attraverso il campo **mod_reg_r/m**; la struttura interna di questo campo viene illustrata dalla Figura 4.

Figura 4 - Campo mod_reg_r/m			
Contenuto	mod	reg	r/m
Posizione bit	7 6	5 4 3	2 1 0

Il sottocampo **reg** occupa **3** bit (posizioni **3**, **4**, **5**) e rappresenta il codice binario del registro a cui fa' riferimento il **direction bit** dell'**Opcode**; con tre bit possiamo rappresentare in totale 2^3**=8** registri differenti. In certi casi, i **3** bit di **reg** vengono combinati con gli **8** bit del campo **Opcode** per formare particolari **Opcode** a **11** bit.
Il sottocampo **mod** occupa **2** bit (posizioni **6**, **7**) e puo' assumere quindi 2^2**=4** valori differenti, e cioe': **00b**, **01b**, **10b**, **11b**; se **mod=11b**, allora anche il secondo operando e' un registro e il suo codice e' rappresentato dai **3** bit del sottocampo **r/m** (posizioni **0**, **1**, **2**). Se invece il sottocampo **mod** e' diverso da **11b**, allora il secondo operando e' una locazione di memoria o un valore immediato (cioe' una costante numerica); se il secondo operando e' una locazione di memoria, allora i due bit di **mod** si combinano con i tre bit di **r/m** per codificare la modalita' di indirizzamento a questo secondo operando. Con **2+3=5** bit possiamo formare 2^5**=32** valori diversi destinati alla codifica del secondo operando; attraverso questi **32** valori possiamo codificare **8** registri (per il caso di **mod=11b**) piu' **24** modalita' di indirizzamento differenti.

Le considerazioni appena svolte ci indicano chiaramente che il passo successivo che dobbiamo compiere, consiste nel procedere alla codifica binaria dei registri della **CPU** e delle varie modalita' di accesso alla memoria; analizziamo prima di tutto la codifica binaria dei registri generali e speciali che viene illustrata dalla Figura 5.

Figura 5 - Codifica dei registri

Registro		**Codice macchina**
w = 0	**w = 1**	
AL	AX	000b
CL	CX	001b
DL	DX	010b
BL	BX	011b
AH	SP	100b
CH	BP	101b
DH	SI	110b
BH	DI	111b

Tenendo conto del fatto che gli operandi di una istruzione possono essere, o tutti a **8** bit, o tutti a **16** bit, vengono utilizzati gli stessi **8** codici per rappresentare sia gli **8** possibili registri a **16** bit, sia gli **8** possibili **half registers** a **8** bit; come gia' sappiamo, grazie al **word bit**, la **CPU** e' in grado di sapere se i codici di Figura 5 si riferiscono ad un registro a **8** bit (**w=0**), oppure ad un registro a **16** bit (**w=1**).

In presenza di un registro di segmento tra gli operandi di una istruzione, vengono impiegati particolari **Opcode**; abbiamo visto in precedenza che nel caso ad esempio del trasferimento dati, l'**Opcode 10001110** indica alla **CPU** che l'operando destinazione e' di tipo **SegReg**. La codifica dei registri di segmento viene effettuata con soli **2** bit come viene illustrato dalla Figura 6; il codice a **2** bit di un **SegReg** viene sistemato sempre nei bit in posizione **3** e **4** del sottocampo **reg** di Figura 4, con il bit in posizione **5** che deve valere **0**.

Figura 6 - Codifica dei registri di segmento

SegReg	Codice macchina
ES	00b
CS	01b
SS	10b
DS	11b

Grazie alla Figura 6, possiamo anche capire come vengono costruiti i codici relativi ai segment override prefix di Figura 2; la struttura di questi prefissi infatti e' rappresentata dagli **8** bit **001_SegReg_110**, dove **SegReg** e' uno dei possibili codici a **2** bit visibili in Figura 6.

Riassumendo quindi, se una istruzione prevede un operando di riferimento di tipo **Reg**, allora il codice macchina a **3** bit di questo registro (ricavato dalla Figura 5) viene inserito nel sottocampo **reg** di Figura 4; inoltre, se **mod=11b**, anche il secondo operando e' un registro, e il suo codice macchina a **3** bit (ricavato dalla Figura 5) viene inserito nel sottocampo **r/m** di Figura 4. In questo caso, per il secondo operando si presentano le **8** possibilita' mostrate in Figura 7.

Figura 7

mod	r/m	Registro	
		w = 0	w = 1
11b	000b	AL	AX
11b	001b	CL	CX
11b	010b	DL	DX
11b	011b	BL	BX
11b	100b	AH	SP
11b	101b	CH	BP
11b	110b	DH	SI
11b	111b	BH	DI

Se invece il sottocampo **mod** e' diverso da **11b**, allora il secondo operando e' di tipo **Mem** o **Imm**; la presenza di un secondo operando di tipo **Imm** viene indicata come al solito dall'**Opcode**. Abbiamo visto infatti che ad esempio, l'**Opcode 1100011w** indica un trasferimento dati da sorgente **Imm** a destinazione **Reg** o **Mem**; se invece il secondo operando e' di tipo **Mem**, bisogna indicare alla **CPU** il metodo utilizzato per accedere alla corrispondente locazione di memoria. La **CPU 8086** fornisce, come e' stato gia' detto, ben **24** modalita' di indirizzamento differenti per accedere ad un operando di tipo **Mem**; tutte queste modalita' indicano naturalmente il metodo di accesso al contenuto di una locazione di memoria individuata da un indirizzo logico del tipo **Seg:Offset**.

Nel precedente capitolo abbiamo gia' incontrato alcuni semplicissimi esempi relativi al modo con cui si accede ad una locazione di memoria; abbiamo visto esempi del tipo **DS:[BX]**, **SS:[BP]**, etc.

La **CPU 8086** permette pero' di rappresentare la componente **Offset** di un indirizzo logico, in modo molto piu' sofisticato; la Figura 8 illustra tutte le **24** modalita' di indirizzamento disponibili.

Figura 8 - Codifica delle modalita' di indirizzamento

mod	r/m	SegReg predefinito	Effective Address
00b	000b	DS	[BX+SI]
00b	001b	DS	[BX+DI]
00b	010b	SS	[BP+SI]
00b	011b	SS	[BP+DI]
00b	100b	DS	[SI]
00b	101b	DS	[DI]
00b	110b	DS	Disp16
00b	111b	DS	[BX]
01b	000b	DS	[BX+SI+Disp8]
01b	001b	DS	[BX+DI+Disp8]
01b	010b	SS	[BP+SI+Disp8]
01b	011b	SS	[BP+DI+Disp8]
01b	100b	DS	[SI+Disp8]
01b	101b	DS	[DI+Disp8]
01b	110b	SS	[BP+Disp8]
01b	111b	DS	[BX+Disp8]
10b	000b	DS	[BX+SI+Disp16]
10b	001b	DS	[BX+DI+Disp16]
10b	010b	SS	[BP+SI+Disp16]
10b	011b	SS	[BP+DI+Disp16]
10b	100b	DS	[SI+Disp16]
10b	101b	DS	[DI+Disp16]
10b	110b	SS	[BP+Disp16]
10b	111b	DS	[BX+Disp16]

In Figura 8, la struttura che puo' assumere la componente **Offset** di un indirizzo logico viene chiamata **Effective Address** (indirizzo effettivo); i termini **Disp8** e **Disp16** indicano un offset rappresentato da un valore esplicito rispettivamente a **8** e a **16** bit. Per ridurre al minimo le dimensioni del codice macchina, l'assembler utilizza, quando e' possibile, solo **8** bit per rappresentare una componente **Disp**; questa situazione si verifica quando **Disp** e' compreso tra **0** e **+127** (cioe' tra **0000h** e **007Fh**).

Osserviamo subito che nel caso piu' semplice, la componente **Offset** dell'indirizzo di memoria a cui vogliamo accedere, viene espressa attraverso un singolo registro puntatore come **BX**, **SI**, etc;

possiamo notare che quando **mod=00b**, non e' presente nessun **Disp** da sommare ad un registro puntatore.

Se vogliamo esprimere in modo piu' sofisticato una componente **Offset**, possiamo servirci di due registri puntatori; in questo caso la componente **Offset** e' data dalla somma degli offset contenuti nei due registri puntatori.
In una situazione di questo genere, il primo registro (quello piu' a sinistra) viene chiamato **registro base**, mentre il secondo registro viene chiamato **registro indice**; nel caso ad esempio di **[BX+SI]**, il registro **BX** e' la **base**, mentre il registro **SI** e' l'**indice**.
In modalita' reale **8086** solamente **BX** e **BP** possono svolgere il ruolo di registro base (da cui il nome **base register** per **BX** e **base pointer register** per **BP**); solamente **SI** e **DI** possono svolgere il ruolo di registri indice (da cui il nome di **index registers** per **SI** e **DI**).
E' proibito usare **BX** e **BP** in coppia; non possiamo scrivere quindi **[BX+BP]** o **[BP+BX]**. Il programmatore puo' accedere in lettura o in scrittura al registro **SP**, ma non puo' dereferenziarlo (non si puo' scrivere cioe' **[SP]**); non e' possibile quindi utilizzare **SP** nelle combinazioni visibili in Figura 8 per esprimere una componente **Offset**.

Come si puo' notare dalla Figura 8, nel caso piu' complesso la componente **Offset** di un indirizzo logico puo' essere espressa con due registri puntatori piu' un ulteriore spiazzamento rappresentato da un offset esplicito a **8** bit (**Disp8**) o **16** bit (**Disp16**); in questo caso la componente **Offset** e' data dalla somma dei contenuti dei due registri puntatori e dello spiazzamento. Queste tre componenti dell'offset vengono chiamate **base**, **indice** e **spiazzamento**; nel caso ad esempio di una componente **Offset** espressa dalla terna **[BX+SI+03F2h]**, il registro **BX** e' la **base**, il registro **SI** e' l'**indice**, mentre **03F2h** e' lo **spiazzamento**.

Nella colonna **SegReg predefinito** di Figura 8, vengono riportati i registri di segmento che la **CPU**, in assenza di segment override, associa automaticamente alla componente **Offset** della colonna **Effective Address**; come si puo' notare, in assenza di **BP** il **SegReg** predefinito e' sempre **DS**. In presenza invece di **BP**, il **SegReg** predefinito e' sempre **SS**; per aggirare le associazioni predefinite visibili in Figura 8, il programmatore deve ricorrere come al solito al segment override esplicito scrivendo ad esempio **SS:[BX+DI+002Eh]**.
Un caso particolarmente interessante per gli **Effective Address** di Figura 8, e' rappresentato dal codice **mod=00b**, **r/m=110b**; come si puo' notare, questo codice macchina viene generato quando la componente **Offset** di un indirizzo di memoria viene espressa da un semplice spiazzamento a **16** bit (come ad esempio **00C8h**). In sostanza, il programmatore ha la possibilita' di specificare un operando di tipo **Mem** attraverso il suo offset esplicito.
Come si puo' facilmente intuire, questa situazione appare piuttosto ambigua, e puo' quindi trarre in inganno l'assembler; se scriviamo ad esempio una istruzione che prevede la somma tra il contenuto di **AX** e il contenuto della locazione di memoria che si trova all'offset esplicito **00C8h** di un segmento di programma, l'assembler crede che la nostra intenzione sia quella di sommare il contenuto di **AX** con il valore immediato **00C8h**!
Per far capire all'assembler che **00C8h** e' una componente **Offset** di un indirizzo logico, dobbiamo specificare esplicitamente anche la relativa componente **Seg**; nel caso allora dell'esempio precedente, dobbiamo dire all'assembler che la somma deve svolgersi tra un operando di tipo **Reg** che e' **AX**, e un operando di tipo **Mem** che e' ad esempio **DS:00C8h**.
Si tenga presente che il simbolo **DS:00C8h** e' perfettamente equivalente al simbolo **DS:[00C8h]** o **[DS:00C8h]**; come viene evidenziato dalla Figura 8, se specifichiamo **DS** come **SegReg** predefinito, l'assembler non inserisce nessun segment override prefix in quando **DS** e' il **SegReg** "naturale" per il segmento dati di un programma.

L'utilizzo di numeri espliciti (come **00C8h**) nei programmi, e' una pratica molto pericolosa; il programmatore infatti puo' facilmente andare incontro ad una svista scrivendo un valore sbagliato.
Per evitare questo problema, anche il linguaggio **Assembly**, come accade con i linguaggi di alto livello, permette l'utilizzo di nomi simbolici per rappresentare i dati di un programma; nel caso dell'esempio precedente, possiamo "etichettare" l'offset **00C8h** con un nome simbolico del tipo **Variabile1**. In questo modo, possiamo dire all'assembler che vogliamo sommare il contenuto di **AX** con il contenuto della locazione di memoria **DS:Variabile1**; in base a cio' che e' stato detto in precedenza, il simbolo **DS:Variabile1** e' perfettamente equivalente al simbolo **DS:[Variabile1]** o **[DS:Variabile1]**.
In relazione alla Figura 8 quindi, il simbolo **Variabile1** non e' altro che una componente **Disp16** di un **Effective Address**; si tratta cioe' di uno spiazzamento esplicito "nascosto" pero' dietro un nome simbolico.

E' necessario sottolineare l'importante differenza che esiste tra una componente **Disp** come **Variabile1** e un registro puntatore come **SI**; supponiamo a tale proposito che all'offset **00C8h** del **Data Segment** di un programma sia presente un dato a **16** bit che vale **3C2Ah**. Utilizziamo **DS** per gestire questo segmento dati, e poniamo **SI=00C8h**; utilizziamo inoltre il nome simbolico **Variabile1** per rappresentare lo stesso offset **00C8h** del segmento dati.
Se ora scriviamo una istruzione che prevede la somma tra **AX** e **SI**, l'assembler genera un codice macchina che dice alla **CPU** di sommare il contenuto di **AX** con il contenuto **00C8h** di **SI**; se invece scriviamo una istruzione che prevede la somma tra **AX** e **[SI]**, l'assembler genera un codice macchina che dice alla **CPU** di sommare il contenuto di **AX** con il contenuto **3C2Ah** della locazione di memoria **DS:[SI]**!
Utilizzando invece **Variabile1** al posto di **SI**, dobbiamo ricordare che non esiste nessuna differenza tra il simbolo **DS:Variabile1** e **DS:[Variabile1]**; in ogni caso quindi, la **CPU** somma il contenuto di **AX** con il contenuto **3C2Ah** della locazione di memoria **DS:[Variabile1]**!

11.2.5 Campo Displacement

Ogni volta che in un **Effective Address** e' presente una componente **Disp**, questa componente viene sistemata nel campo **Displacement** visibile in Figura 1; come si puo' notare dalla stessa Figura 1, il campo **Displacement** puo' essere formato da **0** byte (nessuna componente **Disp**), **1** byte (**Disp8**) o **2** byte (**Disp16**).

11.2.6 Campo Immediate

Se l'operando sorgente di una istruzione e' di tipo **Imm**, allora il valore immediato viene sistemato nel campo **Immediate** visibile in Figura 1; questo campo, se e' presente, e' sempre l'ultimo nel codice macchina di una istruzione. Nel caso di operandi a **8** bit (**w=0**), il campo **Immediate** occupa **1** byte; nel caso invece di operandi a **16** bit (**w=1**), il campo **Immediate** occupa **2** byte.

11.3 Esempi pratici di codici macchina per la CPU 8086

Attraverso le tabelle esposte nella sezione **11.2**, siamo in grado di determinare il codice macchina di qualsiasi istruzione **8086**, dalla piu' semplice alla piu' complessa; a tale proposito, dobbiamo munirci di un documento, come il gia' citato **231455.PDF** (**8086 16 BIT HMOS MICROPROCESSOR**), che illustra la struttura dei campi **Opcode** e **mod_reg_r/m** di tutte le istruzioni appartenenti al set della **CPU 8086**.
In questo capitolo analizzeremo solo alcune semplici istruzioni che vengono largamente utilizzate nei programmi **Assembly**; nei successivi capitoli verranno analizzati i codici macchina che si

riferiscono a numerose altre istruzioni delle **CPU 80x86**.

11.3.1 Addizione tra operandi di tipo Reg/Mem

L'**Opcode 000000dw** indica alla **CPU** che deve essere eseguita una addizione tra operandi di tipo **Reg/Mem**; come gia' sappiamo, sono permesse tutte le combinazioni tra **Reg** e **Mem**, ad eccezione della somma tra **Mem** e **Mem**.

Il caso piu' semplice si presenta quando entrambi gli operandi sono di tipo **Reg**; supponiamo a tale proposito di voler effettuare una somma tra sorgente **AL** e destinazione **BL**.
Osserviamo subito che gli operandi sono a **8** bit, per cui **w=0**; per convenzione, l'operando di riferimento e' il registro destinazione, per cui **d=1**. Ricaviamo quindi:
`Opcode = 00000010b = 02h`
Dalla Figura 5 ricaviamo il codice macchina dell'operando di riferimento **BL=011b**; questo codice va' inserito nel sottocampo **reg** di Figura 4.
Anche il secondo operando (sorgente) e' un registro (**AL**), per cui **mod=11b**; dalla Figura 5 ricaviamo il codice macchina del secondo operando **AL=000b**, che deve essere inserito nel sottocampo **r/m** di Figura 4. Ricaviamo quindi:
`mod_reg_r/m = 11011000b = D8h`
L'istruzione non ha bisogno di altre informazioni, per cui il suo codice macchina generato dall'assembler e':
`00000010b 11011000b = 02h D8h`
Quando la **CPU** incontra questo codice macchina, somma il contenuto di **AL** con il contenuto di **BL**, e mette il risultato in **BL**.

Passiamo alla somma tra sorgente **AX** e destinazione **BX**; nell'**Opcode** l'unico cambiamento riguarda **w=1** per indicare che gli operandi sono a **16** bit. Ricaviamo quindi:
`Opcode = 00000011b = 03h`
In Figura 5 notiamo che i codici macchina di **AX** e **BX** sono gli stessi di **AL** e **BL**, per cui il campo **mod_reg_r/m** rimane invariato; ricaviamo quindi:
`mod_reg_r/m = 11011000b = D8h`
L'istruzione non ha bisogno di altre informazioni, per cui il suo codice macchina generato dall'assembler e':
`00000011b 11011000b = 03h D8h`
Quando la **CPU** incontra questo codice macchina, somma il contenuto di **AX** con il contenuto di **BX**, e mette il risultato in **BX**.

Scambiando di posto **AX** e **BX**, l'**Opcode** rimane invariato in quanto si tratta sempre di una somma **to register** con operandi a **16** bit; ricaviamo quindi:
`Opcode = 00000011b = 03h`
Nel campo **mod_reg_r/m** bisogna naturalmente scambiare di posto i due codici presenti in **reg** e **r/m**; ricaviamo quindi:
`mod_reg_r/m = 11000011b = C3h`
L'istruzione non ha bisogno di altre informazioni, per cui il suo codice macchina generato dall'assembler e':
`00000011b 11000011b = 03h C3h`
Quando la **CPU** incontra questo codice macchina, somma il contenuto di **AX** con il contenuto di **BX**, e mette il risultato in **AX**.

Vediamo ora quello che succede quando uno degli operandi e' di tipo **Mem**; a tale proposito, facciamo riferimento ad un dato a **16** bit che si trova all'offset **002Ah** del **Data Segment** di un programma.
Supponiamo di voler effettuare una somma tra sorgente **Mem** e destinazione **SI**; grazie alla

presenza del registro **SI**, l'assembler rileva che gli operandi sono a **16** bit, per cui **w=1**. L'operando di riferimento e' il registro destinazione, per cui **d=1**; ricaviamo quindi:

```
Opcode = 00000011b = 03h
```

Dalla Figura 5 ricaviamo il codice macchina di **SI=110b**; questo codice viene inserito nel sottocampo **reg** di Figura 4.
Per quanto riguarda l'operando sorgente, supponiamo di volerlo gestire come **Disp16=002Ah**; in questo caso, dalla Figura 8 ricaviamo **mod=00b** e **r/m=110b**. Se utilizziamo **DS** per gestire il **Data Segment**, dobbiamo ricordarci di esprimere l'operando sorgente come **DS:002Ah**, oppure **DS:[002Ah]**, oppure **[DS:002Ah]**; in caso contrario l'assembler crede che vogliamo sommare **SI** con **002Ah**!
La situazione non cambia scrivendo **[002Ah]** in quanto per l'assembler questo simbolo indica il contenuto della costante numerica **002Ah**; ovviamente, il contenuto della costante **002Ah** e' lo stesso valore **002Ah**, e quindi **[002Ah]** equivale a **002Ah**!
Per il campo **mod_reg_r/m** otteniamo quindi:

```
mod_reg_r/m = 00110110b = 36h
```

L'istruzione necessita di una ulteriore informazione che riguarda lo spiazzamento **002Ah** da inserire nel campo **Displacement** di Figura 1; questo valore, pur non superando **007Fh**, non viene convertito dall'assembler in **2Ah** in quanto, come si vede in Figura 8, un semplice **Disp** deve essere obbligatoriamente a **16** bit.
Siccome **DS** e' il **SegReg** naturale per il blocco dati di un programma, l'assembler non inserisce nessun segment override prefix; l'istruzione non ha bisogno di altre informazioni, per cui il suo codice macchina generato dall'assembler e':

```
00000011b 00110110b 0000000000101010b = 03h 36h 002Ah
```

Quando la **CPU** incontra questo codice macchina, somma il contenuto di **SI** con il contenuto di **DS:[002Ah]**, e mette il risultato in **SI**; e' chiaro che prima di far eseguire questa istruzione, il programmatore deve ricordarsi di inizializzare **DS** con la componente **Seg** dell'indirizzo iniziale del **Data Segment**!

Anziche' utilizzare l'offset esplicito **002Ah**, con il rischio di commettere qualche svista, possiamo etichettare questo stesso offset con un nome simbolico come **Variabile1**; in questo caso dobbiamo esprimere l'operando di tipo **Mem** con il simbolo **DS:Variabile1**, oppure **DS:[Variabile1]**, oppure **[DS:Variabile1]**. Il codice macchina che si ottiene e' ovviamente identico; il vantaggio dell'uso di un nome simbolico sta nel fatto che il calcolo del relativo offset viene delegato all'assembler che e' infallibile nello svolgere questo compito.

Cosa succede se il nostro operando sorgente di tipo **Mem** si trova in un segmento di programma gestito con **CS**?
In questo caso l'operando sorgente deve essere espresso come **CS:002Ah** (o **CS:Variabile1**); siccome **CS** non e' il **SegReg** naturale per il **Data Segment** di un programma, l'assembler inserisce all'inizio del codice macchina il segment override prefix **00101110b=2Eh** relativo allo stesso **CS**. Il restante codice macchina rimane invariato, per cui si ottiene:

```
00101110b 00000011b 00110110b 0000000000101010b = 2Eh 03h 36h 002Ah
```

Osserviamo che a causa del segment override prefix, il codice macchina e' cresciuto di un byte; in assenza di questo prefisso, la **CPU** accede a **DS:002Ah** anziche' a **CS:002Ah**!

Nota importante.
Osservando la Figura 8, si nota che non e' previsto nessun codice macchina relativo ad un **effective address** rappresentato da **[BP]**; questo caso particolare viene gestito attraverso **[BP+Disp8]** ponendo **Disp8=00h**. Il corrispondente codice macchina dovra' quindi prevedere anche il campo **Displacement** destinato a contenere il valore **00h** a **8** bit.

Passiamo ora a qualcosa di piu' impegnativo; vogliamo accedere alla locazione di memoria dei precedenti esempi attraverso il registro base **BX**, il registro indice **DI** e uno spiazzamento **Disp8**. Ponendo allora **BX=0010h**, **DI=0010h** e **Disp8=0Ah**, otteniamo:

```
BX + DI + Disp8 = 0010h + 0010h + 0Ah = 002Ah
```

che e' l'offset della locazione di memoria dei precedenti esempi.
Supponiamo ora di voler effettuare una somma tra sorgente **CX** e destinazione **[BX+DI+0Ah]**; come al solito, grazie alla presenza del registro **CX**, l'assembler rileva che gli operandi sono a **16** bit (**w=1**). L'operando di riferimento e' il registro **CX** sorgente, per cui **d=0**; ricaviamo quindi:

```
Opcode = 00000001b = 01h
```

Dalla Figura 5 ricaviamo il codice macchina di **CX=001b**; questo codice viene inserito nel sottocampo **reg** di Figura 4.
Per l'operando destinazione **[BX+DI+Disp8]**, dalla Figura 8 ricaviamo **mod=01b**, **r/m=001b**; il campo **mod_reg_r/m** e' quindi:

```
mod_reg_r/m = 01001001b = 49h
```

L'istruzione necessita inoltre del contenuto di **Disp8=0Ah**; questo valore viene inserito nel campo **Displacement** di Figura 1.
In assenza di segment override, e in presenza di **BX** come registro base, il **SegReg** predefinito e' **DS**, per cui l'assembler non inserisce nessun segment override prefix; e' importante ribadire che in questo caso, il segment override prefix non viene inserito dall'assembler nemmeno se scriviamo in modo esplicito **DS:[BX+DI+0Ah]** (in questo caso infatti, **DS** e' superfluo).
L'istruzione non ha bisogno di altre informazioni, per cui il suo codice macchina generato dall'assembler e':

```
00000001b 01001001b 00001010b = 01h 49h 2Ah
```

Quando la **CPU** incontra questo codice macchina, somma il contenuto di **CX** con il contenuto di **DS:[002Ah]**, e mette il risultato in **DS:[002Ah]**.

Se al posto di **BX** usiamo **BP**, allora **mod=01b**, e **r/m=011b**; il campo **mod_reg_r/m** diventa:

```
mod_reg_r/m = 01001011b = 4Bh
```

In assenza di segment override, e in presenza di **BP** come registro base, il **SegReg** predefinito e' **SS** per cui l'assembler non inserisce nessun segment override prefix; come al solito, il segment override prefix non viene inserito dall'assembler nemmeno se scriviamo in modo esplicito **SS:[BP+DI+0Ah]** (in questo caso infatti, **SS** e' superfluo).
Tutto il resto rimane invariato, per cui il codice macchina generato dall'assembler e':

```
00000001b 01001011b 00001010b = 01h 4Bh 2Ah
```

Quando la **CPU** incontra questo codice macchina, somma il contenuto di **CX** con il contenuto di **SS:[002Ah]**, e mette il risultato in **SS:[002Ah]**.

Se invece l'operando destinazione e' **DS:[BP+DI+0Ah]**, allora l'assembler inserisce il segment override prefix **00111110b=3Eh** relativo a **DS**; in caso contrario, l'offset **[BP+DI+0Ah]** verrebbe riferito a **SS** a causa della associazione predefinita tra **BP** e **SS**. Tutto il resto rimane invariato, per cui il codice macchina generato dall'assembler e':

```
00111110b 00000001b 01001011b 00001010b = 3Eh 01h 4Bh 2Ah
```

Quando la **CPU** incontra questo codice macchina, somma il contenuto di **CX** con il contenuto di **DS:[002Ah]**, e mette il risultato in **DS:[002Ah]**.

11.3.2 Addizione tra sorgente Imm e destinazione Reg/Mem

L'**Opcode 100000sw** indica alla **CPU** che deve essere eseguita una addizione tra sorgente **Imm** e destinazione **Reg/Mem**; in questo caso, il campo **mod_reg_r/m** assume la forma **mod_000_r/m**. I **3** bit del sottocampo **reg** valgono quindi **000b**, e si combinano con gli **8** bit del campo **Opcode** per formare il codice a **11** bit **100000sw000**; questo codice dice appunto alla **CPU** che l'operando sorgente e' di tipo **Imm**, e l'operando destinazione e' di tipo **Reg/Mem**, ed e' codificato dai due sottocampi **mod** e **r/m**.

Notiamo subito che nell'**Opcode**, il **direction bit** viene sostituito dal cosiddetto **sign bit** (bit di segno) indicato con **s**; per capire il perche' di questa situazione, basta osservare innanzi tutto che in un caso di questo genere il bit **d** e' del tutto superfluo in quanto l'operando di tipo **Imm** non puo' che essere la sorgente.
Il bit **d** viene allora sostituito con il bit **s** che in combinazione con il bit **w** indica alla **CPU** se l'operando **Imm** deve essere esteso da **8** a **16** bit con conseguente estensione del bit di segno; naturalmente, l'estensione del bit di segno viene effettuata attraverso le regole gia' esposte in un precedente capitolo. Si possono presentare i seguenti casi:

***** Se **Imm** e' a **8** bit e gli operandi sono a **8** bit, allora **Imm** rimane a **8** bit; si ha quindi **sw=00b**.
***** Se **Imm** e' a **8** bit e gli operandi sono a **16** bit, allora **Imm** viene esteso a **16** bit con conseguente estensione del bit di segno; si ha quindi **sw=11b**.
***** Se **Imm** e' a **16** bit e gli operandi sono a **16** bit, allora **Imm** rimane a **16** bit; si ha quindi **sw=01b**.

Questo accorgimento apparentemente contorto, permette all'assembler di ridurre le dimensioni del codice macchina; vediamo subito alcuni esempi che chiariscono questo aspetto.

Supponiamo di voler effettuare una somma tra sorgente **Imm=+3** e destinazione **DH**; grazie alla presenza di **DH** l'assembler rileva che gli operandi sono a **8** bit, per cui **w=0**. L'operando di tipo **Imm** puo' essere espresso con soli **8** bit (**00000011b**), per cui non deve subire nessuna estensione del bit di segno (**s=0**); otteniamo quindi:
```
Opcode = 10000000b = 80h
```
Come e' stato gia' detto, l'operando destinazione viene codificato attraverso i sottocampi **mod** e **r/m**; in questo caso, la destinazione e' il registro **DH=110b**, per cui **mod=11b** e **r/m=110b**.
Otteniamo quindi:
```
mod_reg_r/m = 11000110b = C6h
```
L'istruzione necessita di una ulteriore informazione che riguarda il valore immediato **+3** (cioe' **00000011b=03h**) da inserire nel campo **Immediate** di Figura 1; in definitiva, il codice macchina generato dall'assembler e':
```
10000000b 11000110b 00000011b = 80h C6h 03h
```
Quando la **CPU** incontra questo codice macchina, somma **00000011b** con il contenuto di **DH** e mette il risultato in **DH**; l'esempio appena illustrato ci permette di constatare che i valori immediati vengono "incorporati" direttamente nel codice macchina del programma.

Rispetto al precedente esempio sostituiamo ora **DH** con **DX**; grazie alla presenza di **DX** l'assembler rileva che gli operandi sono a **16** bit, per cui **w=1**. L'operando di tipo **Imm** puo' essere espresso con soli **8** bit (**00000011b**), per cui deve subire l'estensione del bit di segno (**s=1**); otteniamo quindi:
```
Opcode = 10000011b = 83h
```
Dalla Figura 5 si ricava **DX=010b**, per cui **mod=11b** e **r/m=010b**; otteniamo quindi:
```
mod_reg_r/m = 11000010b = C2h
```
L'istruzione necessita di una ulteriore informazione che riguarda il valore immediato **+3** (cioe' **00000011b=03h**) da inserire nel campo **Immediate** di Figura 1; in definitiva, il codice macchina generato dall'assembler e':
```
10000011b 11000010b 00000011b = 83h C2h 03h
```
Quando la **CPU** incontra questo codice macchina, estende a **16** bit **00000011b** ottenendo **0000000000000011b** che e' appunto la rappresentazione a **16** bit in complemento a **2** di **+3**; successivamente somma questo valore immediato a **DX** e mette il risultato in **DX**.
Possiamo constatare quindi che con questo accorgimento, l'assembler ha risparmiato **1** byte di codice macchina delegando alla **CPU** il compito di estendere l'operando **Imm** da **8** a **16** bit.

Tenendo sempre **DX** come destinazione, poniamo ora **Imm=-3**; grazie alla presenza di **DX** l'assembler rileva che gli operandi sono a **16** bit, per cui **w=1**. L'operando di tipo **Imm** puo' essere

espresso con soli **8** bit (**11111101b**), per cui deve subire l'estensione del bit di segno (**s=1**); il campo **Opcode** e il campo **mod_reg_r/m** restano inalterati rispetto all'esempio precedente.
L'istruzione necessita di una ulteriore informazione che riguarda il valore immediato **-3** (cioe' **11111101b=FDh**) da inserire nel campo **Immediate** di Figura 1; in definitiva, il codice macchina generato dall'assembler e':

```
10000011b 11000010b 11111101b = 83h C2h FDh
```

Quando la **CPU** incontra questo codice macchina, estende a **16** bit **11111101b** ottenendo **1111111111111101b** che e' appunto la rappresentazione a **16** bit in complemento a **2** di **-3**; successivamente somma questo valore immediato a **DX** e mette il risultato in **DX**.

Tenendo sempre **DX** come destinazione, poniamo ora **Imm=-1500**; grazie alla presenza di **DX** l'assembler rileva che gli operandi sono a **16** bit, per cui **w=1**. L'operando di tipo **Imm** richiede almeno **16** bit (**1111101000100100b**), per cui non deve subire nessuna estensione del bit di segno (**s=0**); otteniamo quindi:

```
Opcode = 10000001b = 81h
```

Il campo **mod_reg_r/m** rimane inalterato.
L'istruzione necessita di una ulteriore informazione che riguarda il valore immediato **-1500** (cioe' **1111101000100100b=FA24h**) da inserire nel campo **Immediate** di Figura 1; in definitiva, il codice macchina generato dall'assembler e':

```
10000001b 11000010b 1111101000100100b = 81h C2h FA24h
```

Quando la **CPU** incontra questo codice macchina, somma il valore immediato **FA24h** a **DX** e mette il risultato in **DX**.

Particolare interesse riveste il caso in cui la sorgente e' di tipo **Imm** e la destinazione e' di tipo **Mem**; supponiamo allora di voler sommare la sorgente **Imm=+3800**, con il contenuto di una locazione di memoria che si trova all'offset **00F8h** del **Data Segment** di un programma.
Se esprimiamo la destinazione come **DS:[00F8h]**, possiamo subito constatare che in questo caso l'assembler non e' in grado di ricavare l'ampiezza in bit degli operandi; infatti, **+3800** e' un semplice numero, mentre **DS:[00F8h]** e' un semplice indirizzo di memoria. In una situazione del genere, l'assembler produce un messaggio di errore del tipo:
`Instruction operand must have size`, oppure `Argument needs type override`
Con questo messaggio l'assembler ci sta dicendo che almeno uno degli operandi deve specificare la sua ampiezza in bit!
Per fornire all'assembler questa informazione, possiamo ricorrere a diversi metodi; il metodo piu' diretto consiste nell'utilizzare gli **operatori** mostrati in Figura 9.

Figura 9 - Address size operators

Operatore	Dimensione in bit della locazione di memoria
BYTE PTR	8
WORD PTR	16
DWORD PTR	32
FWORD PTR	48
PWORD PTR	48
QWORD PTR	64
TBYTE PTR	80

Questi operatori vanno applicati ad operandi di tipo **Mem**, e permettono al programmatore di indicare in modo esplicito l'ampiezza in bit della locazione di memoria a cui si vuole accedere.

Tornando al nostro esempio, se vogliamo eseguire una somma tra operandi a **16** bit, dobbiamo esprimere la destinazione come:
`WORD PTR DS:[00F8h]`
Grazie a questa informazione l'assembler pone **w=1**; siccome **+3800** richiede almeno **16** bit (**0000111011011000b**), l'assembler pone anche **s=0**, per cui si ottiene:
`Opcode = 10000001b = 81h`
L'operando destinazione e' di tipo **Disp16**, per cui dalla Figura 8 ricaviamo **mod=00b** e **r/m=110b**; si ottiene quindi:
`mod_reg_r/m = 00000110b = 06h`
Subito dopo il campo **mod_reg_r/m** e' presente il campo **Displacement** nel quale viene inserito l'offset **00F8h** della locazione di memoria; l'istruzione necessita di una ulteriore informazione che riguarda il valore immediato **+3800** (cioe' **0000111011011000b=0ED8h**) da inserire nel campo **Immediate** di Figura 1. Non essendo presente nessun segment override, il codice macchina generato dall'assembler e':
`10000001b 00000110b 0000000011111000b 0000111011011000b = 81h 06h 00F8h 0ED8h`
Quando la **CPU** incontra questo codice macchina, somma il valore immediato **0ED8h** al contenuto a **16** bit della locazione di memoria **DS:[00F8h]**, e mette il risultato in **DS:[00F8h]**.

Supponiamo ora di esprimere l'operando destinazione come:
`WORD PTR ES:[00F8h]`
Rispetto all'esempio precedente, l'unico cambiamento e' dato dalla presenza del segment override **ES**; l'assembler inserisce quindi il prefisso **00100110b=26h** relativo allo stesso **ES**, e ottiene il seguente gigantesco codice macchina:
```
00100110b 10000001b 00000110b 0000000011111000b 0000111011011000b = 26h 81h 06h
00F8h 0ED8h
```
Quando la **CPU** incontra questo codice macchina, somma il valore immediato **0ED8h** al contenuto a **16** bit della locazione di memoria **ES:[00F8h]**, e mette il risultato in **ES:[00F8h]**.
Naturalmente, le considerazioni appena esposte si possono estendere con estrema facilita' al caso di operandi di tipo **Mem** espressi ad esempio come:
`WORD PTR CS:[BX+SI+0CF2h]`

Confrontiamo ora l'enorme codice macchina dell'ultimo esempio, con il codice macchina:
`00000011b 11011000b = 03h D8h`
relativo all'esempio della somma tra sorgente **AX** e destinazione **BX**.
Nel caso della somma tra registri, il codice macchina e' formato da **2** soli byte, che vengono decodificati molto rapidamente dalla **CPU**; a tutto cio' bisogna aggiungere il fatto che la velocita' di accesso agli operandi di tipo **Reg** e' elevatissima.

Nel caso invece dell'ultimo esempio, relativo alla somma tra **Imm** e **Mem** con tanto di segment override, otteniamo un codice macchina da ben **7** byte che richiede un tempo di decodifica piu' alto; in piu' bisogna anche tenere conto del tempo necessario alla **CPU** per accedere in memoria all'operando di tipo **Mem**.

Proprio per questo motivo, nei limiti del possibile il programmatore deve sempre cercare di privilegiare l'utilizzo nelle istruzioni di operandi di tipo **Reg**; in ogni caso, con le moderne **CPU** gli accessi in memoria avvengono sempre piu' velocemente grazie all'uso di accorgimenti come la **cache memory**, la **prefetch queue**, la **pipeline**, etc.

Con le vecchie **CPU** come la **8086** invece, gli accessi in memoria per l'elaborazione dei dati comportavano una grave perdita di tempo; tanto e' vero che nei manuali di questa **CPU** e' sempre presente una tabella come quella mostrata in Figura 10:

Figura 10 - Clock aggiuntivi per gli Effective Address della 8086

Effective Address	EA Clocks
[BX], [SI], [DI], [BP]	5
Disp	6
[BP+DI], [BX+SI]	7
[BP+SI], [BX+DI]	8
[BX+Disp], [SI+Disp], [DI+Disp], [BP+Disp]	9
[BP+DI+Disp], [BX+SI+Disp]	11
[BP+SI+Disp], [BX+DI+Disp]	12

La colonna **EA Clocks** contiene i cicli di clock che bisogna sommare a quelli necessari alla **8086** per eseguire una istruzione che contiene uno degli **effective address** della Figura 10; come si puo' notare, nel caso della presenza di registro base, registro indice e spiazzamento, si puo' arrivare ad un aggravio pari a **12** cicli di clock!

Tornando all'esempio della somma tra **Imm** e **Mem**, se non vogliamo usare gli operatori come **WORD PTR**, possiamo servirci di un altro metodo; questo metodo consiste nell'assegnare a ciascun dato un nome simbolico. Come vedremo nel capitolo successivo, quando l'assembler incontra in una istruzione il nome simbolico di un dato, e' in grado di ricavare l'ampiezza in bit del dato stesso; nel caso allora dell'ultimo esempio, possiamo esprimere l'operando destinazione come **ES:Variabile1** facendo quindi a meno dell'operatore **WORD PTR**.

11.3.3 Operandi di tipo SegReg

Gli esempi presentati in precedenza, coprono la quasi totalita' dei casi che si possono presentare in relazione al tipo di operandi, alle modalita' di indirizzamento delle locazioni di memoria, etc; per analizzare altre situazioni che si possono presentare, vediamo anche qualche esempio relativo al trasferimento dati che e' sicuramente l'istruzione utilizzata in modo piu' massiccio nei programmi **Assembly**.
Cominciamo con un trasferimento dati da **Reg/Mem** a **SegReg**, che ci permette di analizzare il sistema di codifica degli operandi di tipo **SegReg**; come gia' sappiamo, gli operandi di tipo **SegReg** non possono essere utilizzati con l'addizione o con qualsiasi altra operazione logico aritmetica. Nel caso del trasferimento dati, solo uno dei due operandi puo' essere eventualmente di tipo **SegReg**; e' anche proibito il trasferimento dati da **Imm** a **SegReg**.

Nel caso del trasferimento dati da **Reg/Mem** a **SegReg**, l'**Opcode** e' **10001110=8Eh**; il campo **mod_reg_r/m** inoltre, assume la forma **mod_0_SegReg_r/m**.
Per quanto riguarda il campo **Opcode** notiamo che in questo caso **d=1** per indicare che l'operando di riferimento e' il **SegReg** di destinazione; il **word bit** non e' necessario (**w=0**) in quanto gli operandi devono essere necessariamente a **16** bit.
Per quanto riguarda il campo **mod_reg_r/m** notiamo che il sottocampo **reg** diventa **0_SegReg**; ovviamente **SegReg** e' uno dei codici da **2** bit visibili in Figura 6.

Supponiamo di voler trasferire in **ES** il contenuto del registro **DX**; in questo caso, dalla Figura 6 ricaviamo **SegReg=00b**. La sorgente e' il registro **DX=010b**, per cui **mod=11b** e **r/m=010b**;

otteniamo quindi:
```
mod_reg_r/m = 11000010b = C2h
```
L'istruzione non necessita di altre informazioni, per cui il codice macchina generato dall'assembler e':
```
10001110b 11000010b = 8Eh C2h
```
Quando la **CPU** incontra questo codice macchina, copia in **ES** il contenuto di **DX**.

Supponiamo di voler trasferire in **SS** il contenuto della locazione di memoria **CS:[BX+SI+03F8h]**; in questo caso, dalla Figura 6 ricaviamo **SegReg=10b**. La sorgente e' di tipo **[BX+SI+Disp16]**, per cui dalla Figura 8 ricaviamo **mod=10b** e **r/m=000b**; otteniamo quindi:
```
mod_reg_r/m = 10010000b = 90h
```
L'istruzione richiede inoltre il valore **Disp16** pari a **03F8h** da inserire nel campo **Displacement** di Figura 1; l'assembler aggiunge inoltre il prefisso **00101110b=2Eh** per **CS**, e genera il codice macchina:
```
00101110b 10001110b 10010000b 0000001111111000b = 2Eh 8Eh 90h 03F8h
```
Quando la **CPU** incontra questo codice macchina, copia in **SS** il contenuto a **16** bit della locazione di memoria **CS:[BX+SI+03F8h]**.

11.3.4 Espressioni costanti

L'**Assembly** fornisce una serie di operatori che permettono di definire valori numerici di tipo **Displacement** o **Immediate** attraverso complesse espressioni matematiche; per definire ad esempio il valore numerico **+350**, possiamo scrivere:
```
2 + 80 + (100 * 2) + (60 / 3) + (60 - 12)
```
Questa sequenza (volutamente contorta) di operazioni matematiche, ci permette di analizzare le caratteristiche che devono avere queste espressioni; come si puo' notare, gli operandi devono essere tutti dei valori numerici immediati. La divisione (come **60/3**) e' intesa come divisione intera con eventuale troncamento della parte frazionaria del quoziente; il resto quindi sara' nullo solo quando il dividendo e' un multiplo intero del divisore.
Queste espressioni devono avere queste caratteristiche in quanto devono essere valutate e risolte dall'assembler in fase di generazione del codice macchina; nel nostro caso, la precedente espressione viene risolta dall'assembler che ottiene il risultato **350**.
Nota importante.
Come accade per tutti i linguaggi di programmazione di alto livello, anche tra i vari operatori matematici dell'**Assembly** esiste un preciso ordine di valutazione; le moltiplicazioni e le divisioni ad esempio, hanno la precedenza sulle addizioni e sulle sottrazioni. In ogni caso, si sconsiglia vivamente di fare affidamento su queste regole; e' importante che il programmatore faccia sempre uso delle parentesi per indicare in modo chiaro e diretto l'ordine di valutazione.

Gli operatori matematici che l'**Assembly** ci mette a disposizione, si rivelano particolarmente utili anche per gestire i dati di un programma attraverso sofisticati metodi di indirizzamento; supponiamo a tale proposito, che nel **Data Segment** di un programma sia presente una sequenza consecutiva e contigua di **10** dati di tipo **WORD**, con la prima **WORD** che si trova all'offset **002Ah**.
Di conseguenza, la seconda **WORD** viene a trovarsi all'offset **002Ch**, la terza a **002Eh**, la quarta a **0030h** e cosi' via, sino alla decima **WORD** che viene a trovarsi all'offset **003Ch**.
Anziche' assegnare un nome simbolico a ciascun dato, possiamo etichettare solo il primo di essi attraverso ad esempio il nome **VettWord**; questo nome rappresenta quindi la **WORD** che si trova all'offset **002Ah**. A questo punto, utilizzando gli operatori matematici dell'**Assembly**, possiamo dire che:

DS:[VettWord+0] rappresenta la **WORD** all'offset **002Ah**
DS:[VettWord+2] rappresenta la **WORD** all'offset **002Ch**

DS:[VettWord+4] rappresenta la **WORD** all'offset **002Eh**
..
DS:[VettWord+18] rappresenta la **WORD** all'offset **003Ch**

Infatti, **VettWord+0** rappresenta l'offset:
`002Ah + 0000h = 002Ah`
VettWord+2 rappresenta l'offset:
`002Ah + 0002h = 002Ch`
VettWord+4 rappresenta l'offset:
`002Ah + 0004h = 002Eh`
e cosi' via, sino a **VettWord+18** (cioe' **VettWord+12h**) che rappresenta l'offset:
`002Ah + 0012h = 003Ch`

E' anche consentita la sintassi **DS:VettWord[0]**, **DS:VettWord[2]**, **DS:VettWord[4]**, etc.
Vediamo allora quello che succede se vogliamo trasferire in **BX** la **WORD** rappresentata da **DS:VettWord[4]**; l'**Opcode** per il trasferimento dati da **Reg/Mem** a **Reg/Mem** e' **100010dw**.
Il registro destinazione **BX** e' l'operando di riferimento, per cui **d=1**; siccome gli operandi sono a **16** bit (**w=1**), otteniamo:
`Opcode = 10001011b = 8Bh`
Dalla Figura 5 ricaviamo il codice **BX=011b** da inserire nel sottocampo **reg** di Figura 4; per quanto riguarda l'operando sorgente, che e' di tipo **Mem**, l'assembler calcola:
`002Ah + 0004h = 002Eh`
Si tratta in sostanza di un normalissimo **Disp16**, per cui dalla Figura 8 si ricava **mod=00b**, **r/m=110b**; otteniamo allora:
`mod_reg_r/m = 00011110b = 1Eh`
Dopo il campo **mod_reg_r/m** e' presente anche il campo **Displacement** che contiene l'offset **002Eh**; non e' necessario il segment override prefix, per cui l'assembler genera il codice macchina:
`10001011b 00011110b 0000000000101110b = 8Bh 1Eh 002Eh`
Quando la **CPU** incontra questo codice macchina, copia in **BX** la **WORD** presente nella locazione di memoria **DS:[002Eh]**.

11.4 Codice macchina delle istruzioni 80386

Passiamo ora al caso della **80386** che e' la **CPU** di riferimento per le architetture a **32** bit della famiglia **80x86**; tutte le considerazioni che seguono sono quindi valide anche per le **CPU** di classe superiore.
Da questo momento in poi e' necessario distinguere tra la modalita' reale "pura" delle **CPU 8086**, e l'emulazione della modalita' reale **8086** da parte delle **CPU 80386** e superiori; questa distinzione e' necessaria in quanto, come e' stato detto nei precedenti capitoli, un programma che gira in modalita' reale sulle **CPU 80386** o superiori, puo' sfruttare numerose novita' architetturali introdotte a partire dai microprocessori a **32** bit. Abbiamo gia' visto che queste novita' riguardano in particolare: i registri (e quindi anche gli operandi) a **32** bit, i nuovi registri di segmento **FS** e **GS**, etc; appare ovvio il fatto che un programma destinato a girare su una vera **CPU 8086**, non puo' assolutamente utilizzare queste caratteristiche.
Proprio per i motivi appena elencati, nel seguito del capitolo si utilizzera' la definizione: "**modo 16 bit**" per indicare la modalita' reale pura **8086**; la definizione "**modo 32 bit**" verra' invece utilizzata per indicare la modalita' reale **8086** supportata dalle **CPU 80386** e superiori.

Come e' stato ampiamente spiegato nei precedenti capitoli, la **80386** deve garantire la compatibilita' verso il basso per evitare di rendere inutilizzabile l'enorme quantita' di software scritto per le vecchie **CPU 80286** e soprattutto **8086**; in pratica, un qualsiasi programma scritto ad esempio per la **8086**, deve poter girare senza nessuna modifica sulla **80386**.

Per raggiungere questo obiettivo, sulla **80386** viene utilizzato un metodo di codifica delle istruzioni che e' una estensione di cio' che e' stato precedentemente illustrato per la **8086**; questo aspetto viene evidenziato dalla Figura 11 che mostra la struttura generale che assume una istruzione, valida per le **CPU 80386**.

Figura 11 - Struttura di una istruzione 80386

Campo	Dimensione in byte
Instruction Prefix	0 o 1
Address Size Prefix	0 o 1
Operand Size Prefix	0 o 1
Segment Override Prefix	0 o 1
Opcode	1 o 2
mod_reg_r/m	0 o 1
S.I.B.	0 o 1
Displacement	0, 1, 2 o 4
Immediate	0, 1, 2 o 4

Notiamo subito la presenza di due nuovi prefissi opzionali che sono: l'**Address Size Prefix** e l'**Operand Size Prefix**; l'**Address Size Prefix**, se e' presente, occupa **1** byte e dice alla **CPU** che l'accesso ad un eventuale operando di tipo **Mem** avviene attraverso uno spiazzamento a **32** bit. L'**Operand Size Prefix**, se e' presente, occupa **1** byte e dice alla **CPU** che gli operandi dell'istruzione da elaborare hanno una ampiezza di **32** bit; come si puo' facilmente intuire, questi due prefissi svolgono un ruolo fondamentale nella compatibilita' verso il basso delle **CPU 80386**. La Figura 12 illustra l'elenco completo dei codici macchina (in binario e in esadecimale) di tutti i prefissi disponibili.

Figura 12 - Codici macchina dei prefissi

Prefisso		Codice macchina	
		HEX	BIN
Instruction Prefix	LOCK	F0h	11110000b
	REPNE/REPNZ	F2h	11110010b
	REP, REPE/REPZ	F3h	11110011b
Operand Size Prefix		66h	01100110b
Address Size Prefix		67h	01100111b
Segment Override Prefix	ES	26h	00100110b
	CS	2Eh	00101110b
	SS	36h	00110110b
	DS	3Eh	00111110b
	FS	64h	01100100b
	GS	65h	01100101b

Rispetto alla Figura 2, notiamo la presenza dei prefissi di segmento relativi ai due nuovi registri di segmento **FS** e **GS** introdotti proprio a partire dalla **CPU 80386**; notiamo anche che tutti i codici

macchina di Figura 2, rimangono inalterati nella **80386**.

Tornando alla Figura 11, un'altra novita' riguarda il campo **Opcode** che puo' occupare **1** o **2** byte; questo ampliamento e' necessario in quanto gli **8** bit del campo **Opcode** della **8086** non sono sufficienti per codificare tutte le nuove istruzioni disponibili con la **80386**. E' chiaro che tutte le vecchie istruzioni della **8086**, vengono codificate con i soliti **Opcode** da **1** byte; il secondo byte viene invece utilizzato solo per le nuove istruzioni introdotte dalla **80386**.

Il campo **Displacement** puo' assumere una ampiezza di **32** bit necessaria per rappresentare anche gli offset a **32** bit disponibili con la **80386**; osserviamo inoltre che con la **CPU 80386**, il campo **Immediate** puo' contenere ovviamente anche valori numerici a **32** bit.

Il campo **mod_reg_r/m** rimane inalterato; naturalmente, anche con la **80386** in certi casi i **3** bit del sottocampo **reg** si combinano con il campo **Opcode**.

Osserviamo infine la presenza di un campo opzionale da **1** byte chiamato **S.I.B.**; il significato di questo campo viene illustrato nel seguito del capitolo.

In Figura 13 vediamo come vengono codificati i registri generali e speciali sulla **CPU 80386**.

Figura 13 - Codifica dei registri

Modo 16 bit		**Modo 32 bit**		**Codice macchina**
w = 0	**w = 1**	**w = 0**	**w = 1**	
AL	AX	AL	EAX	000b
CL	CX	CL	ECX	001b
DL	DX	DL	EDX	010b
BL	BX	BL	EBX	011b
AH	SP	AH	ESP	100b
CH	BP	CH	EBP	101b
DH	SI	DH	ESI	110b
BH	DI	BH	EDI	111b

Come si puo' notare, nel **modo 16 bit** i codici macchina dei registri sono identici a quelli di Figura 5; per gli operandi di tipo **Reg** a **8** o **16** bit si ottengono quindi codifiche identiche a quelle gia' viste negli esempi per la **CPU 8086**. Se l'**Operand Size Prefix** e' assente, allora **w=0** indica operandi a **8** bit, mentre **w=1** indica operandi a **16** bit; se invece l'**Operand Size Prefix** e' presente, allora **w=0** indica operandi a **8** bit, mentre **w=1** indica operandi a **32** bit.
Come al solito, se l'operando di riferimento e' di tipo **Reg**, allora il suo codice a **3** bit ricavato dalla

Figura 13 viene sistemato nel sottocampo **reg** del campo **mod_reg_r/m**; se anche il secondo operando e' di tipo **Reg**, allora **mod=11b**, mentre **r/m** contiene il codice a **3** bit del registro, ricavato sempre dalla Figura 13.

In Figura 14 vediamo come vengono codificati i registri di segmento sulla **CPU 80386**.

Figura 14 - Codifica dei registri di segmento

SegReg	Codice macchina
ES	000b
CS	001b
SS	010b
DS	011b
FS	100b
GS	101b
riservato	110b
riservato	111b

Abbiamo visto in Figura 6 che nel **modo 16 bit**, i **4** registri di segmento della **8086** vengono codificati con **2** soli bit; nel **modo 32 bit** invece, i **6** registri di segmento vengono codificati con **3** bit. I primi **4** codici di Figura 14 coincidono con quelli di Figura 6, garantendo cosi' la compatibilita' verso il basso; si puo' anche notare che i codici **110b** e **111b** di Figura 14 sono riservati e non devono essere quindi utilizzati per rappresentare un **SegReg** nelle istruzioni.

La **CPU 80386** fornisce inoltre **24** modalita' di indirizzamento a **16** bit per gli operandi di tipo **Mem**; appare ovvio il fatto che le codifiche di queste modalita' di indirizzamento sono assolutamente identiche a quelle mostrate in Figura 8.
Nel **modo 32 bit** invece, sono disponibili **21** modalita' principali di indirizzamento a **32** bit; queste **21** modalita' vengono mostrate dalla Figura 15.

Figura 15 - Codifica delle modalita' di indirizzamento a 32 bit senza S.I.B. byte			
mod	**r/m**	**SegReg predefinito**	**Effective Address**
00b	000b	DS	[EAX]
00b	001b	DS	[ECX]
00b	010b	DS	[EDX]
00b	011b	DS	[EBX]
00b	100b	--	S.I.B. byte
00b	101b	DS	Disp32
00b	110b	DS	[ESI]
00b	111b	DS	[EDI]
01b	000b	DS	[EAX+Disp8]
01b	001b	DS	[ECX+Disp8]
01b	010b	DS	[EDX+Disp8]
01b	011b	DS	[EBX+Disp8]
01b	100b	--	S.I.B. byte
01b	101b	SS	[EBP+Disp8]
01b	110b	DS	[ESI+Disp8]
01b	111b	DS	[EDI+Disp8]
10b	000b	DS	[EAX+Disp32]
10b	001b	DS	[ECX+Disp32]
10b	010b	DS	[EDX+Disp32]
10b	011b	DS	[EBX+Disp32]
10b	100b	--	S.I.B. byte
10b	101b	SS	[EBP+Disp32]
10b	110b	DS	[ESI+Disp32]
10b	111b	DS	[EDI+Disp32]

La Figura 15 ci permette di constatare che nel **modo 32 bit**, tutte le regole di indirizzamento relative alla **8086** vengono completamente stravolte; infatti, qualunque registro a **32** bit della **80386** puo' svolgere il ruolo di registro puntatore. Notiamo che negli **effective address** di Figura 15, non puo' essere utilizzato il registro **ESP**.
Le associazioni predefinite tra registri puntatori e registri di segmento visibili in Figura 15, non presentano nessuna novita' rispetto a quanto si e' visto in Figura 8; possiamo dire quindi che anche con i registri puntatori a **32** bit, in assenza di **EBP**, il **SegReg** predefinito e' sempre **DS**, mentre in presenza di **EBP**, il **SegReg** predefinito e' sempre **SS**.

Alcune combinazioni tra **mod** e **r/m** visibili in Figura 15, indicano la presenza del campo **S.I.B.** nel codice macchina di una istruzione; il significato di questo campo verra' illustrato nel seguito del capitolo.

Nota importante.
Nei precedenti capitoli e' stato sottolineato il fatto che nella modalita' reale del **DOS**, la componente **Offset** di un indirizzo logico deve essere espressa da un valore a **16** bit compreso tra **0000h** e **FFFFh**; di conseguenza, questi stessi limiti devono essere rispettati anche quando si lavora in modalita' reale con gli **effective address** a **32** bit della **80386**. Utilizzando, ad esempio, **EBX** per esprimere una componente **Offset**, dobbiamo fare in modo che il contenuto di questo registro sia compreso tra **00000000h** e **0000FFFFh**; in caso contrario si ottiene un programma che provoca un sicuro crash!
Nella la modalita' protetta di **Windows**, un programma che gira in modalita' reale senza rispettare i limiti appena illustrati per gli spiazzamenti, viene immediatamente interrotto; subito dopo viene mostrata una finestra di errore per informare l'utente che il programma ha tentato di accedere ad un'area riservata della memoria. Questo e' un classico esempio relativo ai meccanismi di protezione disponibili con la modalita' protetta delle **CPU 80386** e superiori.

11.4.1 Confronto tra "modo 16 bit" e "modo 32 bit"

In base alle considerazioni esposte in precedenza, si puo' facilmente intuire che se proviamo a ripetere gli esempi presentati nella sezione **11.3**, riotteniamo gli stessi identici codici macchina; verifichiamolo subito partendo dall'addizione tra sorgente **AL** e destinazione **BL**.
Abbiamo gia' visto che l'**Opcode** per l'addizione tra **Reg/Mem** e **Reg/Mem** e' **000000dw**; naturalmente, questo stesso **Opcode** fa' parte del set di istruzioni dalla **80386**. Abbiamo quindi **w=0** (operandi a **8** bit) e **d=1** (operazione **to register**), per cui:
```
Opcode = 00000010b = 02h
```
Dalla Figura 13 ricaviamo il codice macchina dell'operando di riferimento **BL=011b**, per cui **reg=011b**; sempre dalla Figura 13 ricaviamo il codice macchina del secondo operando **AL=000b**, per cui **mod=11b** e **r/m=000b**. Si ottiene quindi:
```
mod_reg_r/m = 11011000b = D8h
```
L'istruzione non ha bisogno di altre informazioni, per cui il suo codice macchina generato dall'assembler e':
```
00000010b 11011000b = 02h D8h
```
Questo codice macchina e' identico a quello ricavato nell'esempio della sezione **11.3**!

Passiamo alla somma tra sorgente **AX** e destinazione **BX**; nell'**Opcode** l'unico cambiamento riguarda **w=1** per indicare che gli operandi sono a **16** bit. Ricaviamo quindi:
```
Opcode = 00000011b = 03h
```
In Figura 13 notiamo che i codici macchina di **AX** e **BX** sono gli stessi di **AL** e **BL**, per cui il campo **mod_reg_r/m** rimane invariato; ricaviamo quindi:
```
mod_reg_r/m = 11011000b = D8h
```
L'istruzione non ha bisogno di altre informazioni, per cui il suo codice macchina generato dall'assembler e':
```
00000011b 11011000b = 03h D8h
```
Questo codice macchina e' identico a quello ricavato nell'esempio della sezione **11.3**!

Vediamo ora quello che succede quando si esegue una somma tra sorgente **EAX** e destinazione **EBX**; dalla Figura 13 si rileva che i codici macchina di **EAX** e **EBX** sono identici a quelli di **AX** e **BX**, per cui l'unico cambiamento rispetto all'esempio precedente riguarda il fatto che gli operandi

sono a **32** bit. L'assembler inserisce quindi nel codice macchina l'**Operand Size Prefix** di Figura 12, che vale **66h** e dice appunto alla **CPU** che gli operandi sono a **32** bit; si ottiene quindi:
```
01100110b 00000011b 11011000b = 66h 03h D8h
```
E' necessario ribadire che in presenza del prefisso **66h**, **w=0** indica che gli operandi sono a **8** bit, mentre **w=1** indica che gli operandi sono a **32** bit (**full size**); questi semplici esempi mostrano in modo chiaro il metodo che viene usato dalla **CPU 80386** per supportare le istruzioni della **CPU 8086**.

Passiamo ad un trasferimento dati da **Imm8** a **Mem8**; poniamo **Imm8=-120**, e **Mem8=DS:[BX+002Dh]**; per tutte le **CPU** della famiglia **80x86**, l'**Opcode** di questa istruzione e' **1100011w**.
Come al solito, per indicare all'assembler che gli operandi sono a **8** bit, dobbiamo esprimere l'operando destinazione come:
```
BYTE PTR [BX+002Dh]
```
Abbiamo quindi **w=0**, per cui si ottiene:
```
Opcode = 11000110b = C6h
```
L'operando destinazione e' del tipo **[BX+Disp8]**, per cui dalla Figura 8 ricaviamo **mod=01b** e **r/m=111b**; il campo **mod_reg_r/m** assume come al solito la forma **mod_000_r/m**, per cui si ottiene:
```
mod_reg_r/m = 01000111b = 47h
```
Dopo il campo **mod_reg_r/m** e' necessario il campo **Displacement** destinato a contenere lo spiazzamento **002Dh** a **8** bit; infine e' necessario il campo **Immediate** che contiene il valore **-120**, cioe' **10001000b=88h** a **8** bit. Non essendo necessario nessun prefisso di segmento, si ottiene il codice macchina:
```
11000110b 01000111b 00101101b 10001000b = C6h 47h 2Dh 88h
```
Questo codice macchina e' identico per qualsiasi **CPU** della famiglia **80x86**.

Sostituendo **BX** con **EBX**, sempre nel caso di un trasferimento dati a **8** bit, dobbiamo esprimere l'operando destinazione come:
```
BYTE PTR [EBX+002Dh]
```
Abbiamo quindi **w=0**, per cui si ottiene:
```
Opcode = 11000110b = C6h
```
L'operando destinazione e' del tipo **[EBX+Disp8]**, per cui dalla Figura 15 ricaviamo **mod=01b** e **r/m=011b**; per il campo **mod_reg_r/m** si ottiene quindi:
```
mod_reg_r/m = 01000011b = 43h
```
Dopo il campo **mod_reg_r/m** e' necessario il campo **Displacement** destinato a contenere lo spiazzamento **002Dh** a **8** bit; e' necessario poi il campo **Immediate** che contiene il valore **-120**, cioe' **10001000b=88h** a **8** bit. L'assembler inserisce anche il prefisso **67h** di Figura 12 per indicare che gli indirizzamenti sono a **32** bit, e ottiene il codice macchina:
```
01100111b 11000110b 01000011b 00101101b 10001000b = 67h C6h 43h 2Dh 88h
```
Questo codice macchina vale solamente per le **CPU 80386** e superiori.

Analizziamo il caso degli operandi a **16** bit con l'operando destinazione espresso come:
```
WORD PTR [BX+002Dh]
```
Abbiamo quindi **w=1**, per cui si ottiene:
```
Opcode = 11000111b = C7h
```
L'operando destinazione e' del tipo **[BX+Disp8]**, per cui dalla Figura 8 ricaviamo **mod=01b** e **r/m=111b**; per il campo **mod_reg_r/m** si ottiene quindi:
```
mod_reg_r/m = 01000111b = 47h
```
Dopo il campo **mod_reg_r/m** e' necessario il campo **Displacement** destinato a contenere lo spiazzamento **002Dh** a **8** bit; e' necessario poi il campo **Immediate** che contiene il valore **-120**, cioe' **1111111110001000b=FF88h** a **16** bit (con estensione del bit di segno). L'assembler ottiene quindi il codice macchina:
```
11000111b 01000111b 00101101b 1111111110001000b = C7h 47h 2Dh FF88h
```

Questo codice macchina e' identico per qualsiasi **CPU** della famiglia **80x86**.

Sostituendo **BX** con **EBX**, sempre nel caso di un trasferimento dati a **16** bit, dobbiamo esprimere l'operando destinazione come:
```
WORD PTR [EBX+002Dh]
```
Abbiamo quindi **w=1**, per cui si ottiene:
```
Opcode = 11000111b = C7h
```
L'operando destinazione e' del tipo **[EBX+Disp8]**, per cui dalla Figura 15 ricaviamo **mod=01b** e **r/m=011b**; per il campo **mod_reg_r/m** si ottiene quindi:
```
mod_reg_r/m = 01000011b = 43h
```
Dopo il campo **mod_reg_r/m** e' necessario il campo **Displacement** destinato a contenere lo spiazzamento **002Dh** a **8** bit; e' necessario poi il campo **Immediate** che contiene il valore **-120**, cioe' **1111111110001000b=FF88h** a **16** bit. L'assembler inserisce anche il prefisso **67h** di Figura 12 per indicare che gli indirizzamenti sono a **32** bit, e ottiene il codice macchina:
```
01100111b 11000111b 01000011b 00101101b 1111111110001000b = 67h C7h 43h 2Dh
FF88h
```
Questo codice macchina vale solamente per le **CPU 80386** e superiori.

Analizziamo il caso degli operandi a **32** bit con l'operando destinazione espresso come:
```
DWORD PTR [BX+002Dh]
```
Abbiamo quindi **w=1**, per cui si ottiene:
```
Opcode = 11000111b = C7h
```
L'operando destinazione e' del tipo **[BX+Disp8]**, per cui dalla Figura 8 ricaviamo **mod=01b** e **r/m=111b**; per il campo **mod_reg_r/m** si ottiene quindi:
```
mod_reg_r/m = 01000111b = 47h
```
Dopo il campo **mod_reg_r/m** e' necessario il campo **Displacement** destinato a contenere lo spiazzamento **002Dh** a **8** bit; e' necessario poi il campo **Immediate** che contiene il valore **-120**, cioe' **11111111111111111111111110001000b=FFFFFF88h** a **32** bit (con estensione del bit di segno). L'assembler inserisce anche il prefisso **66h** di Figura 12 per indicare che gli operandi sono a **32** bit, e ottiene il codice macchina:
```
01100110b 11000111b 01000111b 00101101b 11111111111111111111111110001000b =
66h C7h 47h 2Dh FFFFFF88h
```
Questo codice macchina vale solamente per le **CPU 80386** e superiori.

Sostituendo **BX** con **EBX**, sempre nel caso di un trasferimento dati a **32** bit, dobbiamo esprimere l'operando destinazione come:
```
DWORD PTR [EBX+002Dh]
```
Abbiamo quindi **w=1**, per cui si ottiene:
```
Opcode = 11000111b = C7h
```
L'operando destinazione e' del tipo **[EBX+Disp8]**, per cui dalla Figura 15 ricaviamo **mod=01b** e **r/m=011b**; per il campo **mod_reg_r/m** si ottiene quindi:
```
mod_reg_r/m = 01000011b = 43h
```
Dopo il campo **mod_reg_r/m** e' necessario il campo **Displacement** destinato a contenere lo spiazzamento **002Dh** a **8** bit; e' necessario poi il campo **Immediate** che contiene il valore **-120**, cioe' **11111111111111111111111110001000b=FFFFFF88h** a **32** bit. L'assembler inserisce anche i prefissi **66h** e **67h** di Figura 12 per indicare che, sia gli operandi, sia gli indirizzamenti sono a **32** bit, e ottiene il codice macchina:
```
01100110b 01100111b 11000111b 01000011b 00101101b
11111111111111111111111110001000b =
66h 67h C7h 43h 2Dh FFFFFF88h
```
Questo codice macchina vale solamente per le **CPU 80386** e superiori.

In quest'ultimo esempio, se proviamo ad esprimere l'operando destinazione come:
```
DWORD PTR SS:[EBX+002Dh]
```

imponiamo all'assembler di inserire anche il prefisso **36h** per il registro di segmento **SS**; si ottiene quindi il codice macchina:

```
00110110b 01100110b 01100111b 11000111b 01000011b 00101101b
11111111111111111111111110001000b =
36h 66h 67h C7h 43h 2Dh FFFFFF88h
```

11.4.2 Il campo S.I.B. - Scale Index Base

Abbiamo visto che la **CPU 8086** permette di gestire gli indirizzi di memoria attraverso **effective address** del tipo:

```
[BX+SI+03F8h]
```

Abbiamo anche visto che in questo caso, **BX** rappresenta la **base** dell'indirizzo, **SI** rappresenta l'**indice**, mentre **03F8h** rappresenta lo **spiazzamento**.
Naturalmente, tutto cio' puo' essere fatto anche con i registri a **32** bit della **CPU 80386**; ricordando che in questa **CPU**, i registri a **32** bit sono tutti puntatori, possiamo scrivere ad esempio:

```
[EAX+EDX+000003F8h]
```

Anche in questo caso, **EAX** rappresenta la **base** dell'indirizzo, **EDX** rappresenta l'**indice**, mentre **000003F8h** rappresenta lo **spiazzamento**.
La **CPU 80386** introduce anche una novita' che permette al programmatore di richiedere la moltiplicazione del registro indice per un fattore che puo' essere **1**, **2**, **4**, **8**; questo fattore prende il nome di **scale factor** (fattore di scala). In questo modo possiamo esprimere quindi indirizzi del tipo:

```
[EAX+(EDX*4)+000003F8h]
```

Le varie componenti di questo indirizzo vengono denominate nell'ordine: **base**, **indice**, **scala** e **spiazzamento**.

Per capire come avviene la codifica binaria di questo tipo di indirizzi, cominciamo con l'osservare che in Figura 15, certe combinazioni dei sottocampi **mod** e **r/m** indicano alla **CPU** che il campo **mod_reg_r/m** e' seguito da un ulteriore campo da **1** byte che prende il nome di **S.I.B.** o **Scale Index Base**; il campo **S.I.B.** contiene proprio la codifica binaria della modalita' di accesso in memoria.
Sempre attraverso la Figura 15, possiamo notare in particolare che la presenza del **S.I.B.** byte viene indicata da **r/m=100b**, e **mod** che puo' valere **00b**, **01b** o **10b**; in una situazione del genere, la **CPU** sa' che dopo il campo **mod_reg_r/m** e' presente il campo **S.I.B.**, il quale assume la struttura mostrata in Figura 16.

Figura 16 - Campo S.I.B.

Contenuto	scale		index			base		
Posizione bit	7	6	5	4	3	2	1	0

Il sottocampo **scale** e' formato da **2** bit (posizioni dalla **6** alla **7**), e codifica il fattore di scala; le **4** possibili codifiche del fattore di scala vengono mostrate in Figura 17.

Figura 17 - Codifica del fattore di scala

scale	Scale Factor
00b	x 1
01b	x 2
10b	x 4
11b	x 8

Il sottocampo **index** di Figura 16 occupa **3** bit (posizioni dalla **3** alla **5**), e codifica il registro a **32** bit che svolge il ruolo di **indice**; il sottocampo **base** di Figura 16 occupa **3** bit (posizioni dalla **0** alla **2**), e codifica il registro a **32** bit che svolge il ruolo di **base**. Questi codici a **3** bit possono essere ricavati dalla quarta e quinta colonna della Figura 13.
Per ogni valore del sottocampo **mod** (**00b**, **01b** o **10b**) sono previste **8** differenti modalita' di indirizzamento; in totale abbiamo quindi **3*8=24** modalita' che vengono illustrate dalla Figura 18. Il termine "**(indice scalato)**" indica simbolicamente il prodotto tra un registro indice e un fattore di scala, come ad esempio (**ESI*8**).

Figura 18 - Codifica delle modalita' di indirizzamento a 32 bit con S.I.B. byte

mod	base	SegReg predefinito	Effective Address
00b	000b	DS	[EAX+(indice scalato)]
00b	001b	DS	[ECX+(indice scalato)]
00b	010b	DS	[EDX+(indice scalato)]
00b	011b	DS	[EBX+(indice scalato)]
00b	100b	SS	[ESP+(indice scalato)]
00b	101b	DS	[Disp32+(indice scalato)]
00b	110b	DS	[ESI+(indice scalato)]
00b	111b	DS	[EDI+(indice scalato)]
01b	000b	DS	[EAX+(indice scalato)+Disp8]
01b	001b	DS	[ECX+(indice scalato)+Disp8]
01b	010b	DS	[EDX+(indice scalato)+Disp8]
01b	011b	DS	[EBX+(indice scalato)+Disp8]
01b	100b	SS	[ESP+(indice scalato)+Disp8]
01b	101b	SS	[EBP+(indice scalato)+Disp8]
01b	110b	DS	[ESI+(indice scalato)+Disp8]
01b	111b	DS	[EDI+(indice scalato)+Disp8]
10b	000b	DS	[EAX+(indice scalato)+Disp32]
10b	001b	DS	[ECX+(indice scalato)+Disp32]
10b	010b	DS	[EDX+(indice scalato)+Disp32]
10b	011b	DS	[EBX+(indice scalato)+Disp32]
10b	100b	SS	[ESP+(indice scalato)+Disp32]
10b	101b	SS	[EBP+(indice scalato)+Disp32]
10b	110b	DS	[ESI+(indice scalato)+Disp32]
10b	111b	DS	[EDI+(indice scalato)+Disp32]

Osserviamo subito che questa volta e' consentita anche la presenza del registro puntatore **ESP** che pero' puo' svolgere esclusivamente il ruolo di registro base; e' proibito quindi scrivere indirizzi del tipo **[EAX+ESP+02h]**. Dalla Figura 13 si rileva che il codice macchina di **ESP** e' **100b**; possiamo dire quindi che nel campo **index** del **S.I.B.** byte di Figura 16, non deve assolutamente comparire il codice macchina **100b**!
Dalla Figura 18 si rileva anche un'altro aspetto fondamentale, relativo al fatto che il ruolo di **base**

viene svolto dal registro a **32** bit che si trova piu' a sinistra; questo aspetto e' molto importante in quanto il **SegReg** predefinito dipende proprio dal registro **base**!
Per chiarire questo delicato aspetto, analizziamo i seguenti due casi:
`[EAX+EBP+08h]` e `[EBP+EAX+08h]`
Nel primo caso, il registro base e' **EAX**, per cui il **SegReg** predefinito e' **DS**; nel secondo caso, il registro base e' **EBP**, per cui il **SegReg** predefinito e' **SS**.

Nota importante.
Il **Borland TASM** rispetta in pieno queste regole, mentre il **Microsoft MASM** presenta un grave bug che continua ad esistere anche nelle versioni piu' recenti di questo assembler; a causa di questo bug, il **MASM** quando lavora in modalita' reale a **32** bit, in assenza del fattore di scala inverte il ruolo del registro **base** e del registro **indice**!
Se scriviamo quindi **[EAX+EBP+1Bh]**, il **MASM** utilizza **SS** come **SegReg** predefinito.
Se scriviamo invece **[EBP+EAX+1Bh]**, il **MASM** utilizza **DS** come **SegReg** predefinito.

Per evitare questo pericolosissimo bug del **MASM**, quando si lavora in modalita' reale a **32** bit e' fondamentale specificare sempre il segment override, anche quando questa informazione e' superflua; dobbiamo scrivere quindi:
DS:[EAX+EBP+1Bh], **SS:[EBP+EAX+1Bh]**, etc.
Questo bug non si verifica invece, ne' quando si specifica in modo esplicito il fattore di scala, ne' quando il **MASM** lavora in modalita' protetta.

Si tenga presente che nel **modo 16 bit**, l'ordine con il quale vengono disposti il registro base e il registro indice non ha nessuna importanza; non esiste nessuna differenza quindi tra:
[BP+SI+028Ch] e **[SI+BP+028Ch]**
In entrambi i casi, **BP** e' la base, mentre **SI** e' l'indice; in assenza quindi di segment override, il **SegReg** predefinito e' in ogni caso **SS**.
Per motivi di chiarezza e di stile, si consiglia comunque di disporre le varie componenti di un **effective address**, nel classico ordine **base**, **indice**, **spiazzamento**.

11.4.3 Esempi pratici per il S.I.B. byte

Vediamo alcuni esempi che chiariscono meglio le considerazioni appena esposte; supponiamo di voler trasferire nel registro **ECX**, il contenuto a **32** bit della locazione di memoria **[EDX+ESI+0Eh]**. Abbiamo gia' visto che l'**Opcode** per questa istruzione e' **100010dw**; nel nostro caso **w=1** (operandi a **32** bit) e **d=1** (operazione **to register ECX**), per cui:
`Opcode = 10001011b = 8Bh`
L'operando destinazione e' **ECX**, per cui dalla Figura 13 si ricava **reg=001b**; l'**effective address** e' del tipo:
`[EDX+(indice scalato)+Disp8]`
per cui (Figura 15 e Figura 18) **mod=01b** e **r/m=100b**. Otteniamo quindi:
`mod_reg_r/m = 01001100b = 4Ch`
Il fattore di scala e' **1** (nessuna moltiplicazione), il registro base e' **EDX**, mentre il registro indice e' **ESI**; dalle Figure 13, 17 e 18 otteniamo allora **scale=00b**, **index=110b**, **base=010b**. Per il campo **S.I.B.** si ha quindi:
`S.I.B. = 00110010b = 32h`
L'ultimo campo e' il **Displacement** che contiene il valore **0Eh** a **8** bit; l'assembler inserisce anche i due prefissi **66h** e **67h** per indicare che gli operandi e gli indirizzi sono entrambi a **32** bit. In definitiva, il codice macchina che si ottiene e':
`01100110b 01100111b 10001011b 01001100b 00110010b 00001110b = 66h 67h 8Bh 4Ch`

```
32h 0Eh
```

Questo codice macchina viene generato correttamente dal **Borland TASM**; il **Microsoft MASM** invece, a causa del bug citato in precedenza, inverte **base** con **index** e produce:

```
S.I.B. = 00010110b = 16h
```

Vediamo ora quello che succede, se l'operando destinazione e' il registro **EBX**, e l'operando sorgente e' la locazione di memoria **[EAX+(EBP*8)+03FEh]**; le parentesi tonde servono solo per chiarezza, e non sono indispensabili in quanto la moltiplicazione ha la precedenza sull'addizione.
Il campo **Opcode** rimane invariato, per cui:

```
Opcode = 10001011b = 8Bh
```

L'operando destinazione e' **EBX**, per cui dalla Figura 13 si ricava **reg=011b**; l'**effective address** e' del tipo:
`[EAX+(indice scalato)+Disp32]` (non e' consentito usare un **Disp16**)
per cui (Figura 15 e Figura 18) **mod=10b** e **r/m=100b**. Otteniamo quindi:

```
mod_reg_r/m = 10011100b = 9Ch
```

Il fattore di scala e' **8**, il registro base e' **EAX**, mentre il registro indice e' **EBP**; dalle Figure 13, 17 e 18 otteniamo allora **scale=11b**, **index=101b**, **base=000b**. Per il campo **S.I.B.** si ha quindi:

```
S.I.B. = 11101000b = E8h
```

L'ultimo campo e' il **Displacement** che contiene il valore **000003FEh** a **32** bit; l'assembler inserisce anche i due prefissi **66h** e **67h** per indicare che gli operandi e gli indirizzi sono entrambi a **32** bit. In definitiva, il codice macchina che si ottiene e':

```
01100110b 01100111b 10001011b 10011100b 11101000b
00000000000000000000001111111110b =
66h 67h 8Bh 9Ch E8h 000003FEh
```

Questo codice macchina viene generato correttamente dal **Borland TASM**; anche il **Microsoft MASM**, in presenza di un fattore di scala diverso da **1**, genera correttamente il codice macchina.
Analizziamo infine un caso che non prevede il **S.I.B.** byte; consideriamo a tale proposito un trasferimento dati a **32** bit da **[ECX+2Dh]** a **EBP**.
Anche in questo caso abbiamo l'**Opcode** che vale **100010dw**, con **w=1** e **d=1**, per cui:

```
Opcode = 10001011b = 8Bh
```

L'operando destinazione e' **EBP**, per cui dalla Figura 13 si ricava **reg=101b**; l'**effective address** e' del tipo:

```
[ECX+Disp8]
```

per cui (Figura 15) **mod=01b** e **r/m=001b**. Otteniamo quindi:

```
mod_reg_r/m = 01101001b = 69h
```

L'ultimo campo e' il **Displacement** che contiene il valore **2Dh** a **8** bit; l'assembler inserisce anche i due prefissi **66h** e **67h** per indicare che gli operandi e gli indirizzi sono entrambi a **32** bit. In definitiva, il codice macchina che si ottiene e':

```
01100110b 01100111b 10001011b 01101001b 00101101b =
66h 67h 8Bh 69h 2Dh
```

Per questo tipo di indirizzamenti il bug del **MASM** non si manifesta.

11.5 Il codice Assembly

In base alle considerazioni esposte in questo capitolo, si puo' facilmente capire che l'idea di programmare un computer in codice macchina, appare assolutamente folle; eppure, i primi elaboratori elettronici, venivano programmati proprio in questo modo!
Il programma veniva scritto in codice macchina ed immesso nel calcolatore attraverso i piu' strani meccanismi; una volta che il calcolatore aveva terminato le elaborazioni, forniva i risultati (sempre in codice macchina), attraverso un dispositivo di output che generalmente era rappresentato dalla telescrivente (l'antenata della stampante). A questo punto, i tecnici analizzavano i risultati forniti dal computer (sotto forma di una marea di **0** e **1**), e se veniva riscontrata qualche stranezza, si provvedeva a ricontrollare tutto il programma alla ricerca di eventuali errori.

Una situazione di questo genere creava enormi problemi, sia ai programmatori, sia agli analisti che avevano il compito di studiare i risultati prodotti dal programma; gli studi e le ricerche intraprese per trovare una soluzione a questi problemi, hanno portato alla nascita del linguaggio **Assembly**!
Lo scopo del linguaggio **Assembly** e' quello di permettere la programmazione a basso livello dei computers attraverso una sintassi evoluta, e quindi facilmente comprensibile dagli esseri umani; questo obiettivo deve essere raggiunto senza compromettere minimamente le doti di potenza, di efficienza e di compattezza che caratterizzano i programmi scritti in codice macchina.
Un programma scritto in **Assembly**, rappresenta il cosiddetto **codice sorgente**; questo codice sorgente, prima di poter essere eseguito dal computer, deve essere quindi convertito in codice macchina. Come gia' sappiamo, lo strumento che si occupa di questa fase di conversione prende il nome di **assembler** o assemblatore; l'assemblatore richiede che le istruzioni che formano un programma, assumano la seguente forma generale:

```
mnemonico destinazione, sorgente
```

Un **mnemonico** e' un nome simbolico che ha lo scopo di richiamare alla mente l'operazione che la **CPU** deve svolgere; possiamo indicare ad esempio l'addizione con il mnemonico **ADD**, la sottrazione con **SUB**, il complemento a **1** con **NOT**, il trasferimento dati con **MOV**, etc.
Subito dopo il **mnemonico** troviamo i due operandi **sorgente** e **destinazione**; gli assembler come il **TASM** e il **MASM**, seguono la cosiddetta **sintassi Intel** che prevede che l'operando destinazione venga disposto alla sinistra dell'operando sorgente (con i due operandi separati da una virgola). Altri assembler (come quello della **AT&T** utilizzato nel mondo **UNIX**), prevedono la convenzione opposta.
Attraverso questa sintassi evoluta, possiamo scrivere ad esempio:

```
MOV EAX, CS:[EBX+(EDX*4)+00002F8Ah]
```

Questa istruzione indica in modo chiaro e semplice, un trasferimento dati a **32** bit dalla locazione di memoria **CS:[EBX+(EDX*4)+00002F8Ah]** al registro **EAX**; analogamente, possiamo scrivere:

```
ADD BYTE PTR [BX+028Eh], +15
```

Questa istruzione indica chiaramente una somma a **8** bit tra il valore immediato **+15** e il contenuto della locazione di memoria **DS:[BX+028Eh]**.
Questi due semplici esempi rendono evidente l'abissale differenza che esiste tra **Assembly** e codice macchina in termini di chiarezza e semplicita'; vediamo un ulteriore esempio piu' articolato.
Supponiamo che nel **Data Segment** di un programma siano presenti tre variabili statiche a **16** bit che possiamo chiamare simbolicamente **Var1**, **Var2** e **Var3**; queste tre variabili si trovano, nell'ordine, agli offset **002Ah**, **002Ch** e **002Eh**. Vogliamo sommare **Var1** con **Var2**, mettendo il risultato finale in **Var3**; impiegando un numero volutamente esagerato di istruzioni, otteniamo la porzione di programma mostrata in Figura 19 (in **Assembly** il punto e virgola delimita l'inizio di un commento che termina non appena si va' accapo).

Figura 19 - Porzione di programma in Assembly

```
MOV      AX, DS:Var1    ; copia Var1 in AX
MOV      BX, DS:Var2    ; copia Var2 in BX
ADD      AX, BX         ; somma AX con BX
MOV      DS:Var3, AX    ; copia la somma in Var3
```

Questo programma presuppone che **DS** sia gia' stato inizializzato con la componente **Seg** dell'indirizzo logico iniziale del **Data Segment**; confrontiamo ora il programma **Assembly** di Figura 19 con il corrispondente programma in codice macchina mostrato dalla Figura 20.

Figura 20 - Porzione di programma in codice macchina

```
A1h 002Ah          ; copia Var1 in AX
8Bh 1Eh 002Ch      ; copia Var2 in BX
03h C3h            ; somma AX con BX
A3h 002Eh          ; copia la somma in Var3
```

Appare evidente il fatto che grazie all'**Assembly**, il programmatore puo' concentrarsi sul programma che deve scrivere, delegando poi all'assembler il compito di gestire la conversione in codice macchina; questo modo di procedere comporta una drastica riduzione del rischio di commettere errori, e una altrettanto drastica riduzione del tempo necessario per scrivere un programma.
Nota importante.
Osservando la Figura 20 si nota che la prima e la terza **MOV** hanno un codice macchina da **3** byte, mentre la seconda **MOV**, del tutto simile alle altre due, ha un codice macchina da **4** byte; questa differenza e' dovuta al fatto che nella prima e nella terza **MOV** e' presente l'accumulatore che, come e' stato spiegato nel precedente capitolo, e' il registro preferenziale della **CPU**.
Le istruzioni che fanno uso dell'accumulatore (**AL**, **AH**, **AX** o **EAX**), hanno un codice macchina piu' compatto; un trasferimento dati da **Disp16** ad **AX** o viceversa, ha un codice macchina formato dall'**Opcode** e dallo stesso **Disp16**. Se al posto dell'accumulatore si usa un altro registro, il codice macchina e' formato dall'**Opcode**, dal campo **mod_reg_r/m** e dal **Disp16**; possiamo dire quindi che in generale, le istruzioni che fanno uso dell'accumulatore vengono elaborate piu' rapidamente dalla **CPU**.

Analizzando la Figura 19 e la Figura 20, si puo' constatare che esiste una corrispondenza biunivoca tra codice **Assembly** e codice macchina; cio' significa che ad ogni istruzione **Assembly** corrisponde una e una sola istruzione in codice macchina, e viceversa.
Questo aspetto rende relativamente semplice il compito dell'assembler, che in sostanza, non fa' altro che applicare le regole che, in parte, sono state esposte in questo capitolo; inoltre, una diretta conseguenza della corrispondenza biunivoca citata in precedenza, e' data dall'esistenza dei cosiddetti **disassembler**. Il disassembler e' uno strumento che svolge il lavoro opposto a quello dell'assembler; il suo scopo quindi e' quello di convertire in **Assembly** un programma scritto in codice macchina.
Come si puo' facilmente intuire, i disassembler vengono largamente utilizzati dagli **hackers** per effettuare il cosiddetto **reverse engineering**; questa tecnica consiste nel "carpire" a scopo di studio, i segreti che si celano nel codice macchina dei programmi eseguibili. Naturalmente, il reverse engineering viene "praticato" anche a scopo di lucro da persone con pochi scrupoli.

Queste considerazioni, unite a cio' che e' stato esposto in questo capitolo, ci fanno capire che un programmatore **Assembly** degno di questo nome non puo' assolutamente fare a meno della conoscenza della programmazione in codice macchina; nei prossimi capitoli vedremo che in certi casi, questa conoscenza si rivela estremamente utile e vantaggiosa.

A questo punto, siamo in grado di capire anche l'enorme differenza che esiste tra un assemblatore e un compilatore; abbiamo appena visto che il compito svolto dall'assemblatore e' relativamente semplice in quanto una qualsiasi istruzione in codice macchina, puo' essere convertita in **Assembly** attraverso un procedimento univoco.
Il compilatore invece, si trova davanti ad istruzioni scritte con un linguaggio di alto livello, molto piu' vicino al linguaggio umano che non a quello del computer; il programma di Figura 19 ad esempio, puo' essere riscritto in linguaggio **Pascal** con la seguente unica istruzione:

```
Var3 := Var1 + Var2;
```

Il compilatore quindi ha il difficile compito di convertire istruzioni complesse, in una sequenza di codici macchina che deve essere la migliore possibile in termini di compattezza e di efficienza; puo' capitare allora che due diversi compilatori, incontrando la stessa istruzione, generino due sequenze diverse di codici macchina. Tutto cio' rende (quasi) proibitiva l'idea di realizzare uno strumento analogo al disassembler, che converte il codice macchina di Figura 20 nel codice **Pascal** della precedente istruzione.

Per quanto riguarda infine il confronto tra compilatori ed interpreti, e' stato gia' detto che entrambi presentano vantaggi e svantaggi. I compilatori leggono il programma da noi scritto (codice sorgente) e lo traducono completamente in codice macchina, in un formato cioe', direttamente eseguibile dalla **CPU** alla massima velocita' possibile; l'aspetto negativo di questa tecnica, e' legata al fatto che il codice macchina generato dal compilatore, dovendo essere autosufficiente, contiene al suo interno una miriade di informazioni aggiuntive che determinano un notevole aumento delle dimensioni del codice stesso.
Le caratteristiche degli interpreti sono diametralmente opposte; l'interprete infatti legge la prima istruzione del codice sorgente, la traduce in codice macchina e la fa' eseguire dalla **CPU**, ripetendo poi lo stesso procedimento con l'istruzione successiva. Le conseguenze negative di questa tecnica sono rappresentate dalla esasperante lentezza nell'esecuzione, e dalla impossibilita' di eseguire un programma in assenza dell'interprete; l'unico aspetto positivo invece, e' dato dalle ridottissime dimensioni di questi particolari programmi.
L'esigenza dei programmatori e' quella di scrivere programmi velocissimi ed efficienti, ed e' per questo motivo che i compilatori dominano la scena; naturalmente, un qualunque linguaggio di programmazione di alto livello, per quanto potente possa essere, non potra' mai competere con l'**Assembly**.

Dopo aver fatto conoscenza con le prime istruzioni **Assembly**, e' arrivato il momento di cominciare a scrivere i primi programmi; questi programmi hanno lo scopo di mostrare attraverso semplici esempi, le caratteristiche delle decine e decine di istruzioni che formano questo linguaggio di programmazione. Naturalmente, prima di compiere questo passo bisogna analizzare in dettaglio la struttura generale che assume un programma **Assembly**; lo scopo dei prossimi capitoli e' proprio quello di mostrare come e' organizzato un programma **Assembly**, come si esegue l'assemblaggio del programma e come viene generato il file in formato eseguibile.

Capitolo 12 - Struttura di un programma Assembly

In questo capitolo vengono illustrati in dettaglio tutti gli strumenti che il linguaggio **Assembly** ci mette a disposizione per permetterci di definire la struttura generale di un programma; in particolare, vedremo come si deve procedere per creare i vari segmenti di programma, e studieremo le caratteristiche delle informazioni (codice, dati e stack) destinate a riempire i segmenti stessi.

Nei precedenti capitoli e' stato detto che un programma destinato ad essere eseguito nella modalita' reale **8086**, deve essere suddiviso in uno o piu' blocchi chiamati **program segments** (segmenti di programma); all'interno di questi segmenti vengono distribuite le informazioni che nel loro insieme formano appunto un programma.
Nel caso piu' semplice e intuitivo, ciascun segmento di programma viene riservato ad un solo tipo di informazioni; possiamo avere quindi, segmenti riservati al solo codice, segmenti riservati ai soli dati statici e segmenti riservati ai soli dati temporanei (stack).
In modalita' protetta, il programmatore puo' assegnare a ciascun segmento di programma particolari attributi che definiscono la modalita' di accesso al segmento stesso; si possono avere in questo modo, segmenti con attributo **readable** (leggibile), **writable** (scrivibile) e **executable** (eseguibile).
Un segmento di codice ad esempio, puo' essere "etichettato" come **executable** e **readable**, per cui, eventuali dati statici definiti al suo interno non possono essere modificati; analogamente, un segmento riservato ai dati statici puo' essere etichettato come **readable** e **writable**, in modo da renderlo accessibile in lettura e in scrittura.
In modalita' reale **8086** invece, un qualunque segmento di programma e' allo stesso tempo, leggibile, scrivibile ed eseguibile; cio' implica che un qualunque segmento di un programma **DOS** puo' contenere al suo interno un misto di codice, dati e stack. Nei prossimi capitoli vedremo che in particolari circostanze, puo' essere conveniente scrivere un programma **Assembly** formato da un unico segmento destinato a contenere qualsiasi tipo di informazione; nel caso generale pero', si puo' ottenere un enorme aumento della flessibilita' e della chiarezza di un programma, suddividendolo in almeno tre segmenti che ci permettono di separare in modo ordinato, il codice, i dati e lo stack.

La filosofia dell'**Assembly** consiste nel lasciare al programmatore la massima liberta' di scelta; l'importante e' che il programmatore stesso sappia esattamente cio' che sta facendo.

12.1 Struttura generale di un segmento di programma

La Figura 1 illustra la struttura generale che assume un segmento di un programma **Assembly**.

Figura 1 - Segmento di programma Assembly

```
NomeSegmento SEGMENT Allineamento Combinazione Dimensione Classe

NomeSegmento ENDS
```

Come possiamo notare, un segmento di programma viene "aperto" da un nome simbolico (**NomeSegmento**), seguito dalla direttiva **SEGMENT**; il segmento di programma viene poi "chiuso" dallo stesso nome simbolico, seguito dalla direttiva **ENDS** (**end of segment**).
In **Assembly**, le direttive sono parole riservate attraverso le quali il programmatore impartisce dei veri e propri ordini all'assembler; in questo modo e' possibile "pilotare" il lavoro che l'assembler stesso deve svolgere.

Un nome simbolico come **NomeSegmento**, viene chiamato **identificatore** in quanto serve per identificare simbolicamente una qualche entita' del programma; nel caso generale, un identificatore e' sempre seguito da una direttiva che specifica il tipo di entita' da identificare. Nel caso di Figura 1 ad esempio, la direttiva **SEGMENT** permette all'assembler di classificare **NomeSegmento** come l'identificatore di un segmento di programma; in assenza di questa direttiva, l'assembler produce un

messaggio di errore per informarci che non e' in grado di capire a cosa si riferisca il nome simbolico **NomeSegmento**.

Se si eccettuano alcuni casi particolari, in generale il nome di un identificatore viene scelto a piacere dal programmatore; le regole da rispettare sono le stesse imposte da tutti i linguaggi di programmazione. Un identificatore e' formato da una sequenza di codici ASCII che deve iniziare obbligatoriamente con una lettera dell'alfabeto; al suo interno l'identificatore puo' contenere un misto di lettere dell'alfabeto e cifre numeriche. E' proibito invece l'uso di spazi, segni di punteggiatura, simboli matematici e altri simboli speciali del codice ASCII; l'unica eccezione riguarda il simbolo **underscore '_'** che e' associato al codice ASCII **95**.
Le regole appena enunciate sono del tutto generali e risultano compatibili quindi con qualsiasi assembler; esistono poi altre regole particolari che variano pero' da assembler ad assembler.

Il linguaggio **Assembly** "puro" e' **case-insensitive**, e questo significa che non distingue tra lettere maiuscole e lettere minuscole; per l'**Assembly** i nomi simbolici **Variabile1**, **VARIABILE1**, **variabile1**, etc, rappresentano tutti lo stesso identificatore. Lo stesso discorso vale per i mnemonici delle varie istruzioni dell'**Assembly**; non c'e' differenza quindi tra **PUSH** e **push** o tra **POP** e **pop**.
La situazione cambia nel momento in cui si deve interfacciare l'**Assembly** con i linguaggi di alto livello; in questo caso infatti, bisogna tenere presente che certi linguaggi come il **Pascal** e il **BASIC** sono **case-insensitive**, mentre altri linguaggi come il **C** sono **case-sensitive**. Come vedremo in un apposito capitolo, per potersi adattare alle convenzioni seguite da questi linguaggi, l'**Assembly** permette al programmatore di abilitare o meno la distinzione tra lettere maiuscole e lettere minuscole negli identificatori; si tenga anche presente che in molti linguaggi di alto livello, il carattere underscore usato negli indentificatori, puo' assumere un significato speciale.

Tornando alla Figura 1, e' necessario sottolineare che per motivi di stile e di chiarezza, e' meglio scegliere per l'identificatore di un segmento di programma, un nome che abbia attinenza con l'uso che si vuole fare del segmento stesso; nel caso ad esempio di un segmento di codice possiamo scegliere dei nomi come **SEGM_CODICE**, **SEGCODE**, **CODESEGM**, etc, mentre nel caso di un segmento di dati possiamo scegliere dei nomi come **SEGM_DATI**, **SEGDATA**, **DATASEGM**, etc. Naturalmente, e' proibito usare per gli identificatori nomi di parole riservate come **SEGMENT**, **ENDS**, **PUSH**, **POP**, etc; si tenga presente che esistono anche nomi predefiniti come **CODE**, **DATA**, **STACK**, **CODESEG**, **DATASEG**, **STACKSEG**, etc, che vengono impiegati quando si interfaccia l'**Assembly** con i linguaggi di alto livello.

Per il programmatore, l'identificatore di un segmento di programma ha un significato importantissimo; ricordiamo a tale proposito che non appena un programma viene caricato in memoria, ad ogni segmento di programma viene assegnato un determinato indirizzo fisico iniziale; a questo indirizzo fisico iniziale corrisponde univocamente un indirizzo logico normalizzato **Seg:Offset**. Ebbene:
l'identificatore di un segmento di programma, rappresenta proprio la componente **Seg** dell'indirizzo logico normalizzato da cui inizia in memoria il segmento stesso!

Supponiamo ad esempio di avere un segmento di programma chiamato **CODESEGM**, che in fase di caricamento in memoria, viene fatto partire dall'indirizzo fisico **0BFA2h**; da questo indirizzo fisico ricaviamo univocamente l'indirizzo logico normalizzato **0BFAh:0002h**. Possiamo dire allora che durante la fase di esecuzione del programma si ha:

```
CODESEGM = 0BFAh
```

Una volta che un programma e' stato caricato in memoria, tutti gli indirizzi **Seg:Offset** assegnati alle varie informazioni statiche (dati statici e istruzioni), rimangono fissi per tutta la fase di esecuzione; questo discorso vale quindi anche per la componente **Seg** assegnata ad ogni segmento di programma. Da queste considerazioni si deduce allora che l'identificatore di un segmento di programma contiene un valore immediato (costante) di tipo intero senza segno a **16** bit; questo identificatore puo' essere impiegato come operando sorgente di tipo **Imm16** nelle istruzioni che coinvolgono operandi a **16** bit. Riprendendo il precedente esempio, possiamo scrivere quindi istruzioni del tipo:
`MOV AX, CODESEGM` (trasferimento dati da **Imm16** a **Reg16**).

Per definire in modo dettagliato tutte le caratteristiche di un segmento di programma, dobbiamo servirci di una serie di cosiddetti **attributi** che, come si vede in Figura 1, formano una lista che deve essere disposta subito dopo la direttiva **SEGMENT**; l'ordine con il quale vengono disposti i vari attributi non ha nessuna importanza. Esaminando gli attributi e il contenuto dei vari segmenti di programma, l'assembler ricava una numerosa serie di informazioni che verranno poi passate al **linker**; il compito del linker e' quello di utilizzare queste informazioni, per rendere materialmente effettive tutte le richieste che abbiamo impartito all'assembler.
Gli attributi da assegnare ad un segmento di programma sono tutti facoltativi; l'assembler provvede ad assegnare un valore predefinito a tutti gli attributi non specificati dal programmatore.
Analizziamo ora in modo approfondito il significato dei vari attributi di un segmento di programma.

12.1.1 L'attributo "Allineamento"

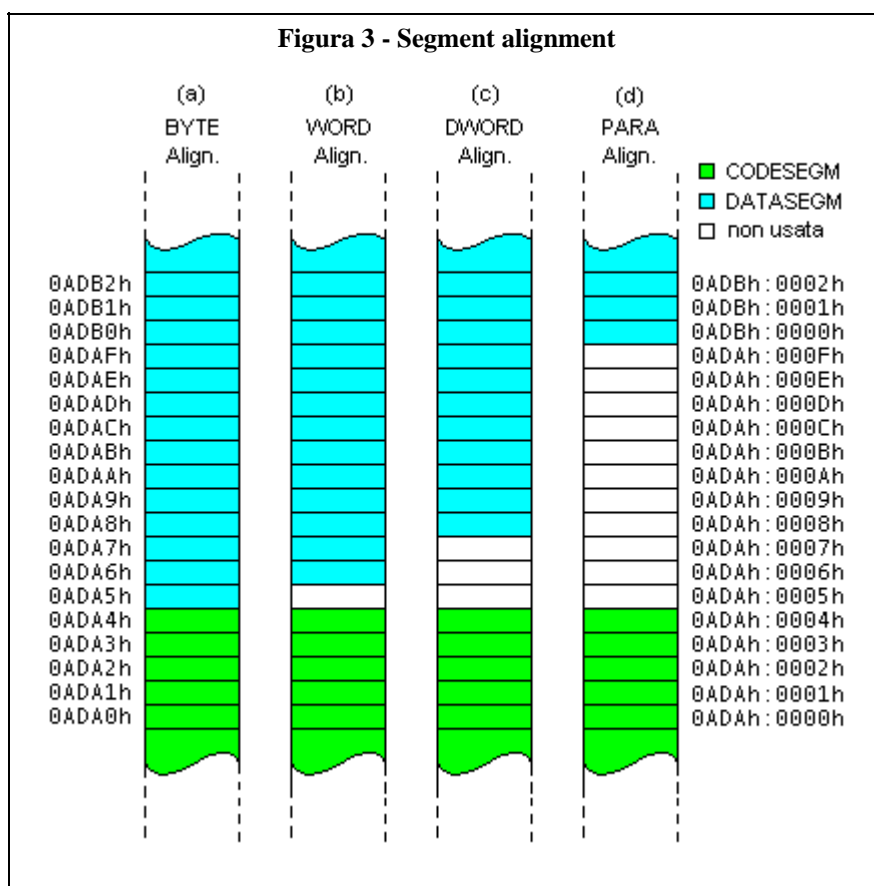
Attraverso l'attributo **Allineamento**, il programmatore puo' richiedere al linker che un determinato segmento di programma venga disposto in memoria a partire da un indirizzo fisico avente particolari requisiti di allineamento; questo aspetto e' molto importante in quanto, come sappiamo, codice e dati correttamente allineati in memoria, vengono acceduti alla massima velocita' possibile dalla **CPU**. Questo attributo opera solo sull'indirizzo fisico iniziale di un segmento di programma, e non sulle informazioni contenute al suo interno; e' chiaro pero' che ad esempio, un segmento dati allineato ad un indirizzo pari, facilita il lavoro del programmatore che ha la necessita' di allineare ad indirizzi pari anche i dati presenti nel segmento stesso.
La tabella di Figura 2 illustra in dettaglio i diversi valori che puo' assumere l'attributo **Allineamento**, e gli effetti prodotti da questi valori sull'indirizzo fisico iniziale di un segmento di programma.

Figura 2 - Attributo "Allineamento"

Allineamento	Indirizzo fisico iniziale del segmento di programma
BYTE	Il prossimo disponibile
WORD	Il prossimo multiplo intero di 2
DWORD	Il prossimo multiplo intero di 4
PARA	Il prossimo multiplo intero di 16
PAGE	Il prossimo multiplo intero di 256
MEMPAGE	Il prossimo multiplo intero di 4096

Per chiarire questi importanti concetti, vediamo subito un esempio pratico; supponiamo a tale proposito di richiedere al **DOS** l'esecuzione di un programma formato nell'ordine, da un blocco codice chiamato **CODESEGM**, da un blocco dati chiamato **DATASEGM** e da un blocco stack chiamato **STACKSEGM**. Come gia' sappiamo, il **DOS** provvede innanzi tutto a trovare un blocco

di memoria le cui dimensioni devono essere sufficienti a contenere il programma da eseguire; una volta che il **DOS** ha caricato il programma in questo blocco di memoria, grazie al lavoro svolto dal linker i vari segmenti si vengono a trovare disposti in modo da rispettare i requisiti di allineamento che noi abbiamo richiesto.
Nel nostro esempio supponiamo che il primo segmento del programma sia **CODESEGM**, e supponiamo anche che questo blocco, dopo il caricamento in memoria, termini esattamente all'indirizzo fisico **0ADA4h**; a questo punto ci interessa sapere come viene disposto il successivo blocco **DATASEGM**.
La Figura 3 illustra proprio il tipo di indirizzo fisico iniziale che viene assegnato a **DATASEGM** in funzione dell'attributo di allineamento che abbiamo scelto per questo segmento di programma; osserviamo che il blocco **CODESEGM** e' stato colorato in verde, mentre il blocco **DATASEGM** e' stato colorato in celeste.

Figura 3 - Segment alignment

Sul lato sinistro della Figura 3 vengono mostrati gli indirizzi fisici a **20** bit associati alle varie celle di memoria; sul lato destro vengono mostrati i corrispondenti indirizzi logici normalizzati. E' importante ribadire che in modalita' reale, ad ogni indirizzo fisico a **20** bit possono corrispondere uno o piu' indirizzi logici generici; ad esempio, dall'indirizzo fisico **0ADA5h** possiamo ricavare gli indirizzi logici **0ADAh:0005h**, **0AD9h:0015h**, **0AD8h:0025h**, etc.
Normalizzando invece gli indirizzi logici, si ottiene una corrispondenza biunivoca con gli indirizzi fisici; in questo caso infatti, all'indirizzo fisico **0ADA5h** corrisponde univocamente l'indirizzo logico normalizzato **0ADAh:0005h**, e viceversa.

In Figura 3a vediamo quello che succede nel caso in cui per **DATASEGM** venga scelto un allineamento di tipo **BYTE**; questa situazione si verifica quando il programmatore apre il blocco **DATASEGM** con una linea del tipo:

```
DATASEGM SEGMENT BYTE
```

In questo caso stiamo richiedendo che **DATASEGM** venga allineato al prossimo indirizzo fisico

libero, successivo al blocco **CODESEGM**; come si puo' notare in Figura 3a, il prossimo indirizzo fisico libero successivo a **0ADA4h** e' ovviamente **0ADA5h**. Il blocco **DATASEGM** viene quindi disposto in memoria a partire dall'indirizzo fisico **0ADA5h**; l'indirizzo logico normalizzato associato a **0ADA5h** e' **0ADAh:0005h**, e cio' significa che:

```
DATASEGM = 0ADAh
```

Possiamo dire quindi che **DATASEGM** viene indirizzato attraverso coppie **Seg:Offset** la cui componente **Seg** vale **0ADAh**; come gia' sappiamo, questa componente **Seg** e' associata all'indirizzo fisico multiplo di **16** (**0ADA0h**) che piu' si avvicina per difetto a **0ADA5h**. Possiamo anche dire che il segmento di programma **DATASEGM** viene inserito nel segmento di memoria n. **0ADAh** a partire dal byte n. **0005h**; di conseguenza, gli offset all'interno di **DATASEGM** partono dal valore minimo **0005h**.
L'indirizzo fisico **0ADA5h** da cui inizia **DATASEGM** e' chiaramente un indirizzo dispari; come si puo' facilmente intuire, questa situazione si rivela adeguata per una **CPU** con architettura a **8** bit, mentre puo' creare problemi di allineamento dei dati nel caso di **CPU** con architettura a **16** bit o superiore.
Osserviamo anche che con l'allineamento di tipo **BYTE** per i segmenti, si ottiene la massima compattezza possibile per un programma; infatti, i vari segmenti vengono disposti in modo consecutivo e contiguo, senza buchi di memoria tra un segmento e l'altro.

In Figura 3b vediamo quello che succede nel caso in cui per **DATASEGM** venga scelto un allineamento di tipo **WORD**; questa situazione si verifica quando il programmatore apre il blocco **DATASEGM** con una linea del tipo:

```
DATASEGM SEGMENT WORD
```

In questo caso stiamo richiedendo che **DATASEGM** venga allineato al prossimo indirizzo fisico libero, successivo al blocco **CODESEGM** e multiplo intero di **2** byte; come si puo' notare in Figura 3b, il prossimo indirizzo fisico libero successivo a **0ADA4h** e multiplo intero di **2** e' ovviamente **0ADA6h**. Il blocco **DATASEGM** viene quindi disposto in memoria a partire dall'indirizzo fisico **0ADA6h**; l'indirizzo logico normalizzato associato a **0ADA6h** e' **0ADAh:0006h**, e cio' significa che:

```
DATASEGM = 0ADAh
```

Possiamo dire quindi che **DATASEGM** viene indirizzato attraverso coppie **Seg:Offset** la cui componente **Seg** vale **0ADAh**; questa componente **Seg** e' associata all'indirizzo fisico multiplo di **16** (**0ADA0h**) che piu' si avvicina per difetto a **0ADA6h**. Possiamo anche dire che il segmento di programma **DATASEGM** viene inserito nel segmento di memoria n. **0ADAh** a partire dal byte n. **0006h**; di conseguenza, gli offset all'interno di **DATASEGM** partono dal valore minimo **0006h**.
L'indirizzo fisico **0ADA6h** da cui inizia **DATASEGM** e' chiaramente un indirizzo pari; come si puo' facilmente intuire, questa situazione si rivela adeguata per tutte le **CPU** con architettura a **16** bit o inferiore, mentre puo' creare problemi di allineamento dei dati nel caso di **CPU** con architettura a **32** bit o superiore.
Osserviamo inoltre che per poter soddisfare i requisiti di allineamento che abbiamo richiesto, viene lasciato un buco di memoria tra **CODESEGM** e **DATASEGM**; questo buco e' pari a **1** byte e rimane inutilizzato.

In Figura 3c vediamo quello che succede nel caso in cui per **DATASEGM** venga scelto un allineamento di tipo **DWORD**; questa situazione si verifica quando il programmatore apre il blocco **DATASEGM** con una linea del tipo:

```
DATASEGM SEGMENT DWORD
```

In questo caso stiamo richiedendo che **DATASEGM** venga allineato al prossimo indirizzo fisico libero, successivo al blocco **CODESEGM** e multiplo intero di **4** byte; come si puo' notare in Figura 3c, il prossimo indirizzo fisico libero successivo a **0ADA4h** e multiplo intero di **4** e' ovviamente **0ADA8h** (infatti, **0ADA6h** pur essendo pari, non e' divisibile per **4**). Il blocco **DATASEGM** viene quindi disposto in memoria a partire dall'indirizzo fisico **0ADA8h**; l'indirizzo logico normalizzato

associato a **0ADA8h** e' **0ADAh:0008h**, e cio' significa che:

```
DATASEGM = 0ADAh
```

Possiamo dire quindi che **DATASEGM** viene indirizzato attraverso coppie **Seg:Offset** la cui componente **Seg** vale **0ADAh**; questa componente **Seg** e' associata all'indirizzo fisico multiplo di **16** (**0ADA0h**) che piu' si avvicina per difetto a **0ADA8h**. Possiamo anche dire che il segmento di programma **DATASEGM** viene inserito nel segmento di memoria n. **0ADAh** a partire dal byte n. **0008h**; di conseguenza, gli offset all'interno di **DATASEGM** partono dal valore minimo **0008h**.
L'indirizzo fisico **0ADA8h** da cui inizia **DATASEGM** e' chiaramente un indirizzo pari multiplo intero di **4**; come si puo' facilmente intuire, questa situazione si rivela adeguata per tutte le **CPU** con architettura a **32** bit o inferiore.
Osserviamo inoltre che per poter soddisfare i requisiti di allineamento che abbiamo richiesto, viene lasciato un buco di memoria tra **CODESEGM** e **DATASEGM**; questo buco e' pari a **3** byte e rimane inutilizzato.

In Figura 3d vediamo quello che succede nel caso in cui per **DATASEGM** venga scelto un allineamento di tipo **PARA**; questa situazione si verifica quando il programmatore apre il blocco **DATASEGM** con una linea del tipo:

```
DATASEGM SEGMENT PARA
```

In questo caso stiamo richiedendo che **DATASEGM** venga allineato al prossimo indirizzo fisico libero, successivo a **CODESEGM** e multiplo intero di **16** byte; come si puo' notare in Figura 3d, il prossimo indirizzo fisico libero successivo a **0ADA4h** e multiplo intero di **16** e' ovviamente **0ADB0h** (ricordiamo infatti, che tutti gli indirizzi fisici multipli di **16**, espressi in esadecimale, hanno sempre la cifra meno significativa che vale **0**). Il blocco **DATASEGM** viene quindi disposto in memoria a partire dall'indirizzo fisico **0ADB0h**; l'indirizzo logico normalizzato associato a **0ADB0h** e' **0ADBh:0000h**, e cio' significa che:

```
DATASEGM = 0ADBh
```

Possiamo dire quindi che **DATASEGM** viene indirizzato attraverso coppie **Seg:Offset** la cui componente **Seg** vale **0ADBh**; questa componente **Seg** e' associata all'indirizzo fisico multiplo di **16** (**0ADB0h**) che piu' si avvicina per difetto a **0ADB0h**. L'allineamento di tipo **PARA** presenta quindi l'importante caratteristica di far coincidere l'inizio di un segmento di programma con l'inizio di un segmento di memoria; nel caso del nostro esempio, l'inizio (**0ADB0h**) del segmento di programma **DATASEGM** coincide con l'inizio del segmento di memoria n. **0ADBh**, e quindi, gli offset all'interno dello stesso **DATASEGM** partono dal valore minimo possibile **0000h**.
L'indirizzo fisico **0ADB0h** da cui inizia **DATASEGM** e' chiaramente un indirizzo pari multiplo intero di **16** (e quindi anche multiplo intero di **4** e di **2**); come si puo' facilmente intuire, questa situazione si rivela adeguata per tutte le **CPU** con architettura a **128** bit o inferiore.
Osserviamo inoltre che per poter soddisfare i requisiti di allineamento che abbiamo richiesto, viene lasciato un buco di memoria tra **CODESEGM** e **DATASEGM**; questo buco e' pari a ben **11** byte e rimane inutilizzato.

Un programmatore **Assembly** degno di questo nome, deve necessariamente conoscere e padroneggiare i concetti appena esposti; come vedremo pero' nel seguito del capitolo, anche i principianti possono stare tranquilli, in quanto tutti questi aspetti, vengono gestiti automaticamente dall'assembler e dal linker.

In modalita' reale, gli attributi di allineamento piu' utilizzati sono **BYTE**, **WORD**, **DWORD** e **PARA**, mentre gli attributi **PAGE** e **MEMPAGE** vengono largamente usati in modalita' protetta; si tenga presente che gli assembler meno recenti potrebbero anche non supportare gli attributi **PAGE** e **MEMPAGE**.
Dalle considerazioni appena esposte risulta chiaramente che in modalita' reale, l'attributo di allineamento piu' adatto per tutte le circostanze e' sicuramente **PARA**, in quanto **16** e' un multiplo intero anche di **4** e di **2** (oltre che ovviamente di **1**); l'unico piccolo difetto dell'attributo **PARA** e'

legato alla eventualita' di un leggero spreco di memoria come risulta evidente anche nell'esempio di Figura 3d.
Nota Importante.
Cio' che e' stato detto in relazione all'attributo di allineamento, ci permette di capire meglio anche la differenza fondamentale che esiste, in modalita' reale **8086**, tra un segmento di memoria e un segmento di programma:
Il segmento di memoria e' un blocco di memoria da 65536 byte, allineato al paragrafo; il segmento di programma e' invece un blocco di memoria che puo' avere un allineamento qualsiasi, e che deve essere interamente contenibile all'interno di un segmento di memoria.
Ne consegue che in modalita' reale **8086**, la dimensione in byte di un segmento di programma non puo' superare la dimensione in byte di un segmento di memoria, e cioe' **65536** byte (**64** Kb).

Se il programmatore definisce un segmento di programma privo dell'attributo di allineamento, **MASM** e **TASM** utilizzano il valore predefinito **PARA**; per motivi di stile e di chiarezza, si consiglia vivamente di specificare sempre tutti gli attributi dei segmenti di programma. In questo modo facilitiamo il compito di chi deve eventualmente occuparsi della manutenzione e dell'aggiornamento del codice sorgente che noi abbiamo scritto; indicando esplicitamente tutti gli attributi di un segmento di programma, evitiamo anche il rischio di scrivere programmi che dipendono dal comportamento predefinito dei diversi assembler.

12.1.2 L'attributo "Combinazione"

Nel caso piu' semplice, un programma **Assembly** e' interamente contenuto all'interno di un unico file; nel seguito del capitolo e nei capitoli successivi, verra' utilizzato il termine **modulo** per indicare ciascuno dei files che contengono il codice sorgente di un programma **Assembly**.
All'interno di un singolo modulo, uno stesso segmento di programma puo' essere aperto e chiuso piu' volte; tutto cio' significa che ad esempio, un segmento aperto dalla linea:

```
DATASEGM SEGMENT WORD
```

dopo essere stato chiuso, puo' essere nuovamente riaperto all'interno dello stesso modulo. Le varie parti che formano **DATASEGM** devono condividere tutte lo stesso nome e gli stessi identici attributi; in fase di assemblaggio del programma, l'assembler provvede a disporre le varie parti una di seguito all'altra in modo da formare un unico segmento di programma. La dimensione in byte del segmento risultante e' pari alla somma delle dimensioni delle varie parti che formano il segmento stesso; naturalmente, questa somma non deve essere superiore a **65536** byte.

La situazione cambia radicalmente nel caso piu' generale di un programma **Assembly** formato da due o piu' moduli; anche in questo caso, il programmatore ha la possibilita' di "spezzare" un segmento di programma in due o piu' parti che possono trovarsi, sia all'interno di uno stesso modulo, sia in moduli differenti.
In una situazione di questo genere, si presenta il problema di come debbano essere combinate tra loro le varie parti che formano un segmento di programma, e che si trovano distribuite in moduli differenti; per poter gestire questa situazione con la massima flessibilita' possibile, l'**Assembly** ci mette a disposizione l'attributo **Combinazione**. Attraverso l'attributo **Combinazione**, il programmatore puo' stabilire se e come il linker debba combinare tra loro i vari segmenti che formano un programma distribuito su due o piu' moduli; la Figura 4 illustra i diversi valori che puo' assumere questo attributo, e gli effetti prodotti da questi valori sui segmenti presenti nel programma.

Figura 4 - Attributo "Combinazione"	
Combinazione	**Effetto**
PRIVATE	Un segmento con attributo PRIVATE non viene combinato con nessun altro segmento presente nel programma; i segmenti di questo tipo restano dunque separati e distinti da tutti gli altri.
PUBLIC	Un segmento con attributo PUBLIC viene unito con tutti gli altri segmenti aventi lo stesso nome e lo stesso attributo PUBLIC, in modo da formare un unico segmento; la dimensione in byte del segmento risultante e' pari alla somma delle dimensioni dei vari segmenti coinvolti nell'unione.
STACK	L'attributo STACK produce gli stessi effetti dell'attributo PUBLIC; in piu', il linker provvede anche ad inizializzare la coppia SS:SP. Il registro SS viene inizializzato con la componente Seg dell'indirizzo logico normalizzato da cui parte il segmento risultante; il registro SP viene inizializzato con la componente Offset che indica la massima capacita' in byte del segmento risultante.
COMMON	Un segmento con attributo COMMON viene unito con tutti gli altri segmenti aventi lo stesso nome e lo stesso attributo COMMON, in modo da formare un unico segmento; i vari segmenti coinvolti nell'unione condividono tutti lo stesso indirizzo fisico iniziale. Cio' significa che questi segmenti vengono sovrapposti tra loro; di conseguenza, la dimensione in byte del segmento risultante e' pari alla dimensione del segmento piu' grande coinvolto nell'unione.
AT XXXXh	Un segmento con attributo AT XXXXh viene fatto partire dall'indirizzo logico XXXXh:0000h, cioe' dall'indirizzo fisico assoluto XXXX0h allineato al paragrafo; questo attributo si rivela particolarmente utile nel momento in cui si ha bisogno di accedere a particolari aree della memoria, come ad esempio, i buffer video, la ROM BIOS, etc.

E' necessario ribadire che l'attributo **Combinazione** opera solo sui segmenti di programma distribuiti su due o piu' moduli; questo attributo non ha quindi nessun effetto su un segmento di programma suddiviso in due o piu' parti disposte tutte all'interno dello stesso modulo.

Per analizzare i vari casi che si possono presentare, supponiamo di avere un programma **Assembly** distribuito in due moduli chiamati **MAIN.ASM** e **LIB1.ASM**; possiamo disporre ad esempio in **MAIN.ASM** il programma principale, e in **LIB1.ASM** una libreria di sottoprogrammi chiamabili dallo stesso programma principale.
Supponiamo ora che in entrambi i moduli, sia presente un segmento di programma aperto da una linea del tipo:

```
DATIPRIVATI SEGMENT WORD PRIVATE
```

In questo caso, i due segmenti **DATIPRIVATI**, nonostante abbiano lo stesso nome e gli stessi attributi, verranno tenuti separati dal linker; cio' e' dovuto alla presenza dell'attributo **PRIVATE**. Il segmento **DATIPRIVATI** presente in **LIB1.ASM** risulta invisibile nel modulo **MAIN.ASM** e viceversa; questo discorso vale anche per i due identificatori **DATIPRIVATI** che, essendo definiti in due moduli differenti, non entrano in conflitto tra loro.
Se i due segmenti **DATIPRIVATI** vengono definiti all'interno dello stesso modulo, vengono uniti tra loro dall'assembler nonostante la presenza dell'attributo **PRIVATE**; in questo modo al linker verra' passato un unico segmento di programma **DATIPRIVATI**.

Supponiamo ora che in entrambi i moduli, sia presente un segmento di programma aperto da una linea del tipo:

```
DATIPUBBLICI SEGMENT WORD PUBLIC
```

In questo caso, i due segmenti **DATIPUBBLICI** vengono uniti tra loro dal linker in modo da formare un unico segmento di programma; cio' e' dovuto alla presenza dell'attributo **PUBLIC**.
I due segmenti di programma possono anche differire tra loro per l'attributo **Allineamento**; possiamo assegnare ad esempio, al primo segmento un allineamento **WORD**, e al secondo segmento un allineamento **PARA**. Il linker provvede ad unire i due segmenti rispettando anche gli attributi di allineamento che abbiamo richiesto per ciascun segmento.

L'attributo **STACK** e' del tutto simile a **PUBLIC**; la differenza fondamentale sta nel fatto che in presenza di un segmento di programma con attributo **STACK**, il linker provvede anche ad inizializzare la coppia **SS:SP**.
Supponiamo che nei due moduli del nostro esempio siano presenti due segmenti aperti con una linea del tipo:

```
STACKSEGM SEGMENT PARA STACK
```

Supponiamo anche che il primo segmento abbia una dimensione di **32** byte, e che il secondo segmento abbia una dimensione di **48** byte; questi due segmenti vengono uniti dal linker che ottiene cosi' un unico segmento **STACKSEGM** grande **32+48=80** byte (cioe' **0050h** byte). Grazie infatti all'allineamento **PARA**, e alle dimensioni (**32** e **64**) multiple di **16**, i due segmenti vengono disposti dal linker in modo consecutivo e contiguo; in sostanza, tra un segmento e l'altro non e' presente nessun buco di memoria.
A questo punto il linker inizializza **SS:SP** ponendo **SP=0050h** (massima capacita' in byte del segmento risultante); per **SS** il linker utilizza un valore simbolico che verra' poi "aggiustato" dal **DOS**. E' chiaro infatti che solo al momento di caricare il programma in memoria sara' possibile conoscere la vera componente **Seg** assegnata a **STACKSEGM**; appare anche intuitivo il fatto che i segmenti con attributo **STACK** si rivelano particolarmente utili per la creazione dello **stack segment** di un programma.
Nel caso del **MASM** si sconsiglia vivamente la creazione di dati statici inizializzati nei segmenti di tipo **STACK** distribuiti su due o piu' moduli; tutti questi argomenti verranno ripresi in modo approfondito nel seguito del capitolo e nei capitoli successivi.

Supponiamo ora che in entrambi i moduli **MAIN.ASM** e **LIB1.ASM**, sia presente un segmento di programma aperto da una linea del tipo:

```
DATICOMUNI SEGMENT DWORD COMMON
```

In questo caso, i due segmenti **DATICOMUNI** vengono sovrapposti tra loro dal linker in modo che condividano entrambi lo stesso indirizzo fisico iniziale (che in questo caso e' allineato alla **DWORD**); il segmento risultante ha quindi una dimensione in byte pari a quella del segmento piu' grande coinvolto nella sovrapposizione.
Questa situazione implica che le informazioni contenute nel primo segmento **DATICOMUNI**, verranno sovrascritte (tutte o in parte) dalle informazioni contenute nel secondo segmento **DATICOMUNI** che viene sovrapposto al primo; bisogna prestare quindi particolare attenzione quando si definiscono dati inizializzati in questo tipo di segmenti.
Supponiamo ad esempio che il primo segmento contenga **35** byte di dati, e che il secondo segmento contenga **20** byte di dati; subito dopo la sovrapposizione, si ottiene un segmento risultante con i primi **20** byte del primo segmento che vengono sovrascritti dai **20** byte del secondo segmento.
Osserviamo anche che la dimensione del segmento risultante e' pari a quella del primo segmento, e cioe' **35** byte.
I segmenti di tipo **COMMON** vengono utilizzati ad esempio dal **BASIC** per la condivisione dei dati tra due o piu' moduli; di conseguenza, come vedremo in un apposito capitolo, i segmenti **COMMON** sono necessari anche per la condivisione dei dati tra moduli **BASIC** e moduli **Assembly**.

Analizziamo infine i segmenti con attributo **AT XXXXh**; come e' stato gia' spiegato, attraverso questo potente attributo, il programmatore ha la possibilita' di posizionare un segmento di programma direttamente all'indirizzo logico iniziale **XXXXh:0000h**, cioe' all'indirizzo fisico assoluto **XXXX0h** allineato al paragrafo.
Ricordando ad esempio che la **ROM BIOS** del computer viene mappata in **RAM** in una finestra da **64** Kb che parte dall'indirizzo logico **F000h:0000h**, possiamo accedere a questa finestra attraverso un apposito segmento di programma aperto da una linea del tipo:

```
BIOSDATA SEGMENT AT 0F000h
```

(il perche' dello zero alla sinistra di **F000h** verra' spiegato piu' avanti).
Per i segmenti di tipo **AT XXXXh**, gli altri attributi in genere vengono omessi; in caso contrario, il **MASM** richiede che **AT XXXXh** sia l'ultimo attributo della lista.
Non e' consentito definire dati inizializzati in un segmento di programma di tipo **AT XXXXh**; in caso contrario, l'assembler genera un semplice messaggio di avvertimento per informarci che le varie inizializzazioni verranno ignorate.

Se il programmatore definisce un segmento di programma privo dell'attributo **Combinazione**, il **MASM** e il **TASM** si servono dell'attributo predefinito **PRIVATE**.

12.1.3 L'attributo "Dimensione"

L'attributo **Dimensione** si riferisce all'ampiezza in bit delle componenti **Offset** utilizzate per indirizzare un segmento di programma; la Figura 5 illustra i due valori disponibili per questo attributo.

Figura 5 - Attributo "Dimensione"

Dimensione	Ampiezza Offset
USE16	16 bit
USE32	32 bit

Come gia' sappiamo, in modalita' reale gli offset devono essere compresi tra **0000h** e **FFFFh**, per cui dobbiamo utilizzare l'attributo **USE16**; possiamo definire ad esempio un segmento del tipo:

```
DATASEGM SEGMENT WORD PUBLIC USE16
```

Questo tipo di segmenti devono avere ovviamente una ampiezza massima di **64** Kb in quanto, con gli offset a **16** bit non possiamo accedere ad informazioni che si trovano oltre questo limite.
In modalita' protetta possiamo assegnare ad un segmento di programma l'attributo **USE16** (offset a **16** bit) o **USE32** (offset a **32** bit); con gli offset a **32** bit possiamo indirizzare teoricamente segmenti da $\mathbf{2^{32}=4}$ Gb!

Se il programmatore definisce un segmento di programma privo dell'attributo **Dimensione**, il **MASM** e il **TASM** si servono di un attributo predefinito che dipende dalla eventuale presenza di apposite direttive che specificano il set di istruzioni che vogliamo utilizzare; in assenza di queste direttive, **MASM** e **TASM** assumono che il programmatore voglia utilizzare il set di istruzioni della **8086**, per cui l'attributo **Dimensione** predefinito e' **USE16**.

Alternativamente possiamo indicare in modo esplicito le direttive illustrate in Figura 6; questa figura mostra anche l'ampiezza predefinita per gli offset, che **MASM** e **TASM** utilizzano in funzione del set di istruzioni che abbiamo specificato.

Figura 6 - Direttiva "Processor"		
Direttiva	**Istruzioni utilizzate**	**Dimensione predefinita**
.8086	Set della CPU 8086	USE16
.186	Set della CPU 80186	USE16
.286	Set della CPU 80286	USE16
.386	Set della CPU 80386	USE32
.486	Set della CPU 80486	USE32
.586	Set della CPU 80586 (classe Pentium I)	USE32
.686	Set della CPU 80686 (classe Pentium II)	USE32

Queste direttive vengono inserite in genere all'inizio di ogni modulo, e hanno il solo scopo di specificare il set di istruzioni che vogliamo utilizzare; e' chiaro quindi che un pessimo programma che utilizza la direttiva **.8086**, non migliora di certo con la direttiva **.586**!
Un programma con direttiva **.8086** gira su qualsiasi **CPU** di classe **8086** o superiore, mentre un programma con direttiva **.586** non puo' girare su **CPU** di classe **80486** o inferiore; si tenga anche presente che esistono numerose altre direttive **Processor** per le **CPU** e le **FPU**, che verranno illustrate nei capitoli successivi.

Nota importante.

Osservando la Figura 6, appare evidente il fatto che per evitare sorprese, il programmatore dovrebbe specificare sempre l'attributo **Dimensione** per ogni segmento di programma; questo attributo ha una importanza enorme in quanto l'assembler lo utilizza per stabilire automaticamente la modalita' predefinita di indirizzamento.
Nell'assemblare un segmento di programma dotato di attributo **USE16**, l'assembler assegna ad ogni informazione un indirizzo logico **Seg:Offset** da **16+16** bit; nell'assemblare un segmento di programma dotato di attributo **USE32**, l'assembler assegna ad ogni informazione un indirizzo logico **Seg:Offset** da **16+32** bit (in questo caso quindi, la componente **Offset** e' un valore a **32** bit).
Come e' stato spiegato in precedenza, in modalita' reale e' necessario creare segmenti di programma dotati di attributo **USE16**; utilizzando l'attributo **USE32**, si ottengono programmi il cui codice macchina e' incompatibile con la modalita' di indirizzamento reale (in fase di esecuzione, un programma di questo genere provoca un sicuro crash)!

12.1.4 L'attributo "Classe"

Attraverso l'attributo **Classe**, il programmatore ha la possibilita' di imporre al linker l'ordine con il quale disporre i vari segmenti che formano un programma; questo attributo e' formato da una stringa racchiusa tra apici singoli. Nel mondo dell'**Assembly** vengono largamente utilizzate le classi consigliate dal **MASM**, come ad esempio '**DATA**', '**CODE**', '**STACK**', etc; queste classi hanno il pregio di indicare chiaramente il tipo di informazioni contenute in un segmento di programma. Le classi **MASM** diventano obbligatorie quando si interfaccia l'**Assembly** con i linguaggi di alto livello; se invece dobbiamo scrivere un programma in puro **Assembly**, nessuno ci impedisce di utilizzare classi del tipo '**MELA**', '**CILIEGIA**', '**LIMONE**', etc.

Per capire il principio di funzionamento dell'attributo **Classe**, bisogna dire innanzi tutto che il linker esamina i vari segmenti di programma nello stesso ordine con il quale sono stati disposti nel codice

sorgente dal programmatore; supponiamo allora che il nostro programma sia distribuito nei due moduli **MAIN.ASM** e **LIB1.ASM**. Nel modulo **MAIN.ASM** sono presenti i seguenti segmenti di programma:

```
LOCALDATA SEGMENT PARA PRIVATE USE16 'DATA'

DATASEGM SEGMENT PARA PUBLIC USE16 'DATA'

CODESEGM SEGMENT PARA PUBLIC USE16 'CODE'

STACKSEGM SEGMENT PARA STACK USE16 'STACK'
```

Nel modulo **LIB1.ASM** sono presenti i seguenti segmenti di programma:

```
CODESEGM SEGMENT PARA PUBLIC USE16 'CODE'

LOCALDATA SEGMENT PARA PRIVATE USE16 'DATA'

DATASEGM SEGMENT PARA PUBLIC USE16 'DATA'

STACKSEGM SEGMENT PARA STACK USE16 'STACK'
```

Il linker parte dal modulo **MAIN.ASM**, e incontra per primo il segmento **LOCALDATA** di classe **'DATA'**; questo segmento e' **PRIVATE**, per cui non verra' combinato con altri eventuali segmenti **LOCALDATA** presenti nel modulo **LIB1.ASM**.
Siccome **LOCALDATA** e' di classe **'DATA'**, il linker va' a cercare tutti gli altri segmenti di classe **'DATA'**; nel modulo **LIB1.ASM** il linker aveva gia' incontrato un secondo segmento **LOCALDATA**, che essendo **PRIVATE** non viene combinato con il precedente **LOCALDATA**.
Nel modulo **MAIN.ASM** viene incontrato **DATASEGM** che viene unito invece con il **DATASEGM** del modulo **LIB1.ASM**.
Non essendoci altri segmenti di classe **'DATA'**, il linker passa al segmento **CODESEGM** di classe **'CODE'** presente nel modulo **MAIN.ASM**; questo segmento viene unito con il segmento **CODESEGM** presente in **LIB1.ASM**.
Non essendoci altri segmenti di classe **'CODE'**, il linker passa al segmento **STACKSEGM** di classe **'STACK'** presente nel modulo **MAIN.ASM**; questo segmento viene unito con il segmento **STACKSEGM** presente in **LIB1.ASM**.
Alla fine il linker genera un programma eseguibile la cui struttura interna e' la seguente:

```
LOCALDATA SEGMENT PARA PRIVATE USE16 'DATA'

LOCALDATA SEGMENT PARA PRIVATE USE16 'DATA'

DATASEGM SEGMENT PARA PUBLIC USE16 'DATA'

CODESEGM SEGMENT PARA PUBLIC USE16 'CODE'

STACKSEGM SEGMENT PARA STACK USE16 'STACK'
```

Come possiamo notare, il linker ha ordinato i vari segmenti disponendo, prima quelli di classe **'DATA'**, poi quelli di classe **'CODE'**, e infine quelli di classe **'STACK'**.
E' importante anche osservare che l'attributo **Classe**, se e' presente, diventa determinante nella fase di combinazione dei vari segmenti di programma; due segmenti **PUBLIC** ad esempio, vengono combinati tra loro solo se hanno lo stesso nome e la stessa classe!

Il metodo di ordinamento dei segmenti di programma puo' essere gestito anche attraverso le direttive illustrate in Figura 7.

Figura 7 - Direttive per l'ordinamento dei segmenti

Direttiva	Tipo ordinamento
.SEQ	Sequenziale
.ALPHA	Alfabetico
.DOSSEG	Standard DOS

La direttiva **.SEQ** (predefinita) indica al linker che i segmenti di programma devono essere disposti nello stesso ordine stabilito dal programmatore nel codice sorgente; in presenza dell'attributo **Classe**, i vari segmenti vengono ordinati sequenzialmente, e raggruppati per classi.
La direttiva **.ALPHA** indica al linker che i segmenti di programma devono essere disposti in ordine alfabetico rispetto ai loro nomi; in presenza dell'attributo **Classe**, i vari segmenti vengono ordinati alfabeticamente, e raggruppati per classi.
La direttiva **.DOSSEG** indica al linker che i segmenti di programma devono essere disposti secondo lo standard **DOS**, e cioe', prima i segmenti di codice, poi i segmenti di dati statici, e infine lo stack; la direttiva **.DOSSEG** e' obsoleta e non dovrebbe essere piu' utilizzata.

Esiste infine un metodo diretto che permette al programmatore di indicare al linker il tipo di ordinamento dei segmenti di programma; questo metodo e' basato sull'uso dei cosiddetti **dummy segments** (letteralmente, **segmenti fantoccio**). Si tratta di segmenti di programma vuoti che hanno il solo scopo di imporre al linker la sequenza di ordinamento dei segmenti stessi; riprendendo il precedente esempio relativo ai due moduli **MAIN.ASM** e **LIB1.ASM**, proprio all'inizio del modulo **MAIN.ASM** possiamo disporre i **dummy segments** mostrati in Figura 8.

Figura 8 - Dummy segments

```
; modulo main.asm

; dummy segments

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'
CODESEGM    ENDS

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'
DATASEGM    ENDS

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'
STACKSEGM   ENDS

; altri segmenti di programma
.............................
```

Siccome il linker incontra per primi questi tre **dummy segments**, alla fine genera un programma eseguibile la cui struttura interna e' la seguente:

```
CODESEGM SEGMENT PARA PUBLIC USE16 'CODE'

DATASEGM SEGMENT PARA PUBLIC USE16 'DATA'

LOCALDATA SEGMENT PARA PRIVATE USE16 'DATA'

LOCALDATA SEGMENT PARA PRIVATE USE16 'DATA'
```

```
STACKSEGM SEGMENT PARA STACK USE16 'STACK'
```

Come si puo' notare, questa volta i segmenti di programma vengono ordinati ponendo per primi quelli di classe **'CODE'**, seguiti poi da quelli di classe **'DATA'** e infine da quelli di classe **'STACK'**; si tenga presente che in determinate circostanze, l'ordinamento dei segmenti di un programma **Assembly** assume una importanza enorme.

12.1.5 Considerazioni finali sui segmenti di programma

Per chiudere questa prima parte del capitolo, si possono riassumere alcuni concetti importanti sui segmenti di programma.
In modalita' reale **8086**, un segmento di programma non puo' contenere piu' di **65536** byte di informazioni (**64** Kb); se abbiamo bisogno ad esempio di piu' di **64** Kb di dati, dobbiamo ripartirli in due o piu' segmenti di dati. Lo stesso discorso vale per i segmenti di codice e per i segmenti di stack; in relazione allo stack bisogna dire che generalmente un programma ha bisogno al massimo di qualche migliaio di byte per i dati temporanei, per cui un solo segmento di stack e' piu' che sufficiente (la gestione di un programma con due o piu' segmenti di stack e' piuttosto impegnativa e richiede una notevole padronanza del linguaggio **Assembly**).
Le considerazioni appena esposte si applicano anche alla combinazione dei segmenti di programma attraverso gli attributi illustrati in Figura 4; tutti i segmenti che scaturiscono da queste combinazioni, devono avere una dimensione complessiva che non puo' superare i **64** Kb.

Non ci si deve preoccupare se i concetti sin qui esposti appaiono ancora poco chiari; tutto cio' che riguarda i segmenti di programma e il loro contenuto, verra' compreso meglio attraverso svariati esempi pratici che verranno presentati nel seguito del capitolo e nei capitoli successivi.

Dopo aver esaminato in dettaglio le caratteristiche generali dei segmenti di programma, possiamo occuparci ora di tutto cio' che riguarda il contenuto dei segmenti stessi; vediamo quindi come bisogna procedere per inserire codice, dati e stack nei segmenti che formano un programma **Assembly**.

12.2 Definizione dei dati statici elementari dell'Assembly

L'**Assembly**, piu' che un linguaggio di programmazione, e' un insieme di strumenti attraverso i quali si puo' fare praticamente di tutto; per essere precisi, bisogna dire che l'**Assembly** e' uno strato software intermedio, attraverso il quale il programmatore puo' accedere in modo molto semplice ai potenti e complessi strumenti forniti dalla **CPU**. Lavorare in **Assembly** significa quindi dialogare direttamente con il mondo binario della **CPU**; tutto cio' si ripercuote anche sul procedimento che bisogna seguire per definire i dati statici di un programma **Assembly**.

Abbiamo gia' visto che i **dati statici** sono cosi' definiti, in quanto esistono staticamente per tutta la fase di esecuzione di un programma; in sostanza, a ciascun dato statico di un programma, viene assegnata una locazione fissa di memoria che permane per tutta la fase di esecuzione del programma stesso. Nei linguaggi di alto livello, i dati statici sono quelli che vengono definiti al di fuori di qualsiasi sottoprogramma (procedura o funzione); come vedremo in un capitolo successivo, i dati definiti all'interno di un sottoprogramma vengono in genere sistemati nello stack in modo che siano distrutti quando il sottoprogramma stesso termina.
Quando si lavora con i linguaggi di alto livello, si ha l'impressione che i relativi compilatori ed

interpreti siano in grado di distinguere tra diversi tipi di dati; la Figura 9 ad esempio, illustra una serie di definizioni di dati statici in un programma scritto in linguaggio **C**.

Figura 9 - Dati statici del C

```
signed char    c1 = 32;        /* intero con segno a 8 bit */
signed int     i1 = -1280;     /* intero con segno a 16 bit */
unsigned int   i2 = 39800;     /* intero senza segno a 16 bit */
unsigned int   vi[10];         /* vettore di 10 interi senza segno a 16 bit */
float          f1 = 3.14;      /* reale con segno a 32 bit */
char           vstr[] = "Stringa alfanumerica";
..........................
```

Come si puo' notare in Figura 9, sembrerebbe che i compilatori **C** siano in grado di distinguere tra numeri interi con o senza segno, numeri con la virgola (reali), stringhe alfanumeriche, etc; in realta', i compilatori e gli interpreti **C**, **Pascal**, **FORTRAN**, **BASIC**, etc, non fanno altro che simulare via software la distinzione che effettivamente esiste tra i diversi tipi di dati.
Nel momento in cui il codice sorgente viene tradotto in codice macchina, questa distinzione scompare; come gia' sappiamo infatti, la **CPU** e' in grado di maneggiare esclusivamente numeri binari, e non ha la piu' pallida idea di cosa sia un numero reale, una stringa, una lettera dell'alfabeto, etc.
In relazione alla fase di definizione di un qualsiasi dato statico, la **CPU** ha bisogno di conoscere tre informazioni fondamentali che sono:
*** l'offset**
*** l'ampiezza in bit**
*** il contenuto iniziale**

Supponiamo ad esempio di avere un segmento dati chiamato **DATASEGM**; all'offset **002Dh** di questo segmento e' presente un dato statico da **16** bit, che contiene il valore iniziale **3AC8h**. Le tre informazioni fondamentali associate a questo dato sono quindi, l'offset **002Dh**, l'ampiezza in bit **16** e il contenuto iniziale **3AC8h**.
Il programmatore **Assembly** deve attenersi a queste esigenze della **CPU**; cio' significa che per ogni dato statico che definiamo in un programma, dobbiamo specificare in modo chiaro l'offset, l'ampiezza in bit e il valore iniziale. A tale proposito, dobbiamo servirci di una apposita sintassi che assume la seguente forma:

```
NomeSimbolico DIRETTIVA valore_iniziale
```

Il **NomeSimbolico** viene scelto a piacere dal programmatore, e deve presentare le caratteristiche gia' discusse in relazione ai nomi dei segmenti di programma; questo nome e' facoltativo, e il suo scopo e' quello di permettere al programmatore di individuare con estrema facilita' l'offset del dato a cui il nome stesso fa' riferimento. Come gia' sappiamo infatti, l'assembler tratta **NomeSimbolico** come un **Disp16** attraverso il quale si puo' accedere al contenuto della corrispondente locazione di memoria; ogni volta che l'assembler incontra **NomeSimbolico** in una istruzione, calcola automaticamente il relativo **Disp16** da inserire nel codice macchina dell'istruzione stessa.

Il **NomeSimbolico** e' seguito da una **DIRETTIVA** che specifica l'ampiezza in bit del dato che stiamo creando; le direttive disponibili con le **CPU 80x86** vengono illustrate dalla Figura 10.

Figura 10 - Direttive Assembly per i dati statici

Nome	Direttiva	Ampiezza (bit)	Valore	
			min.	max.
Define Byte	DB	8	0	255
Define Word	DW	16	0	65535
Define Double word	DD	32	0	4294967295
Define Far pointer	DF	48	0	2^{48}-1
Define far Pointer	DP	48	0	2^{48}-1
Define Quad word	DQ	64	0	2^{64}-1
Define Ten byte	DT	80	0	2^{80}-1

Attraverso le direttive di Figura 10, poste all'interno di un qualsiasi segmento di programma, possiamo chiedere all'assembler di creare locazioni di memoria di ampiezza specificata, riservate alle variabili statiche del nostro programma; in queste locazioni di memoria, possiamo inserire generici numeri binari compresi tra i limiti **min.** e **max.** specificati dalla stessa Figura 10.

Le direttive di Figura 10 sono in genere seguite da un valore immediato che prende il nome di **valore inizializzante**; consideriamo ad esempio la seguente linea posta all'interno di un segmento di programma chiamato **DATASEGM**:

```
Variabile1 DW 2F8Dh
```

Quando l'assembler incontra questa linea all'interno di **DATASEGM**, capisce che in quel preciso punto, cioe' in quel preciso offset interno a **DATASEGM**, deve riservare uno spazio pari a **16** bit da riempire con il valore iniziale **2F8Dh**. Nel momento in cui il programma viene caricato in memoria, in corrispondenza dell'offset interno a **DATASEGM**, individuato dal nome **Variabile1**, sara' presente una locazione da **16** bit contenente il valore iniziale **2F8Dh**; questa locazione potra' quindi essere utilizzata nelle istruzioni, come operando di tipo **Mem**.
Ricordando le cose dette nel precedente capitolo, possiamo anche scrivere:

```
PesoNetto DW (2 + 181) - 4 + (6 * (10 / 5))
```

L'importante e' che l'espressione inizializzante sia formata esclusivamente da valori immediati; questa espressione infatti, deve essere risolta dall'assembler in fase di assemblaggio del programma.

Se non vogliamo specificare un valore iniziale, possiamo scrivere:

```
Variabile1 DW ?
```

In questo caso, il contenuto iniziale di **Variabile1** e' casuale o, come si dice in gergo, "**sporco**"; in sostanza, quando l'assembler incontra il simbolo **'?'** associato alla definizione di **Variabile1**, non effettua nessuna inizializzazione della corrispondente locazione di memoria.

Le direttive **DF** e **DP** vengono utilizzate in modalita' protetta per creare indirizzi **FAR Seg:Offset** a **48** bit, con componente **Seg** a **16** bit e componente **Offset** a **32** bit; in modalita' reale, queste due direttive definiscono semplicemente locazioni di memoria da **48** bit.

Osservando la Figura 10 possiamo notare che e' possibile definire dati aventi una ampiezza in bit maggiore dell'architettura della **CPU**; anche con una **80386** ad esempio, possiamo scrivere:

```
Var64 DQ 3D668AC1FF283AB2h
```

Nell'ipotesi che **Var64** si trovi all'offset **00F2h** di un segmento di programma gestito con **DS**, si ottiene per questa variabile la disposizione in memoria mostrata in Figura 11.

Figura 11 - Var64 in memoria

Contenuto	3Dh	66h	8Ah	C1h	FFh	28h	3Ah	B2h
Offset	00F9h	00F8h	00F7h	00F6h	00F5h	00F4h	00F3h	00F2h

Naturalmente, in questo caso dobbiamo ricordarci che la **80386** puo' maneggiare via hardware numeri binari formati al massimo da **32** bit; di conseguenza, se vogliamo utilizzare **Var64** in una istruzione, dobbiamo "spezzare" questo dato in tante parti da **8**, da **16** o da **32** bit.
Utilizzando ad esempio gli **address size operators** mostrati nel precedente capitolo, possiamo scrivere istruzioni del tipo:

```
MOV EAX, DWORD PTR DS:Var64[0]
```

Osservando la Figura 11 possiamo notare che l'operando sorgente di questa istruzione fa' chiaramente riferimento ai primi **32** bit della locazione di memoria che si trova all'offset:

```
00F2h + 0000h = 00F2h
```

In questo caso vengono quindi copiati in **EAX** i **32** bit meno significativi di **Var64** (**EAX=FF283AB2h**).
Analogamente, possiamo scrivere l'istruzione:

```
MOV EBX, DWORD PTR DS:Var64[4]
```

Osservando la Figura 11 possiamo notare che l'operando sorgente di questa istruzione fa' chiaramente riferimento ai primi **32** bit della locazione di memoria che si trova all'offset:

```
00F2h + 0004h = 00F6h
```

In questo caso vengono quindi copiati in **EBX** i **32** bit piu' significativi di **Var64** (**EBX=3D668AC1h**).

12.2.1 Formati IEEE per i numeri in floating point

Gli assembler piu' sofisticati come **MASM** e **TASM**, permettono di utilizzare persino un numero reale come valore inizializzante; in questo caso sono permesse solo le tre direttive **DD**, **DQ** e **DT**. E' possibile scrivere ad esempio:

```
PiGreco DD 3.14
```

Quando l'assembler incontra una definizione del genere, converte il numero reale **3.14** in una apposita codifica binaria a **32** bit; a titolo di curiosita', il numero reale **3.14** viene codificato a **32** bit come **01000000010010001111010111000011b**.
Il sistema di codifica binaria dei cosiddetti **floating point numbers** (numeri reali in virgola mobile), e' stato stabilito dall'**IEEE** (**Institute of Electrical and Electronics Engineers**); la Figura 12 illustra i tre formati previsti per questi numeri, e le relative caratteristiche (ampiezza in bit, limite inferiore, limite superiore, numero di cifre significative dopo la virgola).

Figura 12 - Formati IEEE per i numeri in floating point

Formato	**bit**	**Direttiva**	**Valore**		**Cifre significative**
			min.	**max.**	
short real	32	DD	1.18×10^{-38}	3.40×10^{38}	6-7
long real	64	DQ	2.23×10^{-308}	1.79×10^{308}	15-16
temporary real	80	DT	3.37×10^{-4932}	1.18×10^{4932}	19

I tre formati **IEEE** si riferiscono tutti a numeri reali con segno e sono stati adottati ormai da tutti i linguaggi di programmazione; la Figura 12 mostra i limiti inferiore e superiore dei numeri reali positivi; per ottenere i corrispondenti limiti inferiore e superiore dei numeri reali negativi, basta mettere il segno meno davanti ai valori **min.** e **max.** di Figura 12.

Per poter lavorare seriamente con i numeri reali, e' necessario disporre di una apposita **FPU** o **Fast**

Processing Unit (unita' di elaborazione rapida); la **FPU** e' in grado infatti di operare via hardware direttamente sui numeri reali, eseguendo su di essi, ad altissima velocita', complessi calcoli matematici che coinvolgono, funzioni logaritmiche, trigonometriche, esponenziali, etc.
La **CPU** invece opera esclusivamente su valori binari che codificano numeri interi con o senza segno; su questi numeri la **CPU** puo' solo eseguire operazioni elementari come addizioni, sottrazioni, negazioni, scorrimenti dei bit, etc. Tutto cio' significa che se definiamo il dato **PiGreco** mostrato in precedenza, la **CPU** lo trattera' come un normalissimo numero binario a **32** bit; se proprio vogliamo operare sui numeri reali con una semplice **CPU**, dobbiamo procedere via software scrivendo complesse procedure per la simulazione dei numeri in floating point.
Tutte le **CPU 80486 DX** e superiori, sono dotate di **FPU** incorporata; con le vecchie **CPU** invece, era necessario acquistare a parte una apposita **FPU** che veniva identificata dalle sigle **8087**, **80187**, **80287**, etc.
Tutti gli aspetti relativi alla **FPU** e ai numeri in virgola mobile, verranno analizzati nella sezione **Assembly Avanzato**.

12.2.2 Nuove direttive MASM per i dati statici

Le direttive mostrate in Figura 10 sono standard, e sono riconosciute quindi da tutti gli assembler destinati al mondo delle **CPU 80x86**; se abbiamo la necessita' di scrivere programmi compatibili con diversi assembler, o se abbiamo a disposizione un vecchio assembler, dobbiamo necessariamente servirci delle direttive standard di Figura 10.
Il **MASM** ha introdotto da tempo una serie di direttive alternative, che permettono di definire i vari formati di dati, utilizzando una sintassi molto simile a quella dei linguaggi di alto livello; queste nuove direttive vengono anche supportate dalle versioni piu' recenti del **TASM** (versione **5.0** o superiore). La Figura 13 illustra la sintassi di queste nuove direttive **MASM**; per i floating point vengono mostrati in questa figura solo i limiti **min.** e **max.** positivi.

Figura 13 - Nuove direttive MASM per i dati statici

Formati di base (generici)				
Direttiva	**Equivalente**	**bit**	**Valore**	
			min.	**max.**
BYTE	DB	8	0	255
WORD	DW	16	0	65535
DWORD	DD	32	0	4294967295
FWORD	DF o DP	48	0	2^{48}-1
QWORD	DQ	64	0	2^{64}-1
TBYTE	DT	80	0	2^{80}-1
Interi con segno				
Direttiva	**Equivalente**	**bit**	**Valore**	
			min.	**max.**
SBYTE	DB	8	-128	+127
SWORD	DW	16	-32768	+32767
SDWORD	DD	32	-2147483648	+2147483647
Numeri reali (Floating-point)				
Direttiva	**Equivalente**	**bit**	**Valore**	
			min.	**max.**
REAL4	DD	32	1.18×10^{-38}	3.40×10^{38}
REAL8	DQ	64	2.23×10^{-308}	1.79×10^{308}
REAL10	DT	80	3.37×10^{-4932}	1.18×10^{4932}

Queste nuove direttive aiutano il programmatore a scrivere programmi piu' comprensibili, in quanto rendono evidente l'uso che vogliamo fare di un determinato dato; possiamo scrivere ad esempio:

```
Dislivello SBYTE -75
```

E' chiaro pero' che indipendentemente dall'aspetto formale, la sostanza non cambia; dal punto di vista della **CPU** infatti, **Dislivello** indica una locazione di memoria da **8** bit che contiene il valore binario iniziale **10110101b** (cioe' **181**); questo valore e' la rappresentazione a **8** bit in complemento a **2** del numero negativo **-75**.
Per indicare a chi legge il nostro programma, che vogliamo interpretare **10110101b** come **-75**, utilizziamo la direttiva **SBYTE**; se invece vogliamo interpretare **10110101b** come **+181**, possiamo

utilizzare la direttiva **BYTE**. Questa distinzione quindi e' puramente formale; la sostanza e' che in entrambi i casi, il contenuto binario di **Dislivello** e' **10110101b**.
Lo stesso discorso vale naturalmente per i numeri reali definiti con le direttive **REAL4**, **REAL8** e **REAL10**; questi numeri vengono visti dalla **CPU** come generici numeri binari costituiti da una sequenza rispettivamente di **32**, **64** e **80** bit. A dimostrazione del fatto che l'unico tipo di dato gestibile dalla **CPU** e' quello intero, possiamo tranquillamente sommare tra loro un **DWORD** con un **REAL4** senza che l'assembler generi alcun messaggio di errore; per la **CPU** infatti, sia il **DWORD** che il **REAL4** sono generici numeri binari a **32** bit. Tutto cio' non sarebbe possibile con i linguaggi di alto livello che simulano via software uno stretto controllo sui tipi di dati; il linguaggio **Pascal** ad esempio, impedisce al programmatore di effettuare una somma tra un **Single** (reale con segno a **32** bit) e un **Longint** (intero con segno a **32** bit).

In sostanza, possiamo dire che tutti i dati definiti con le direttive di Figura 13, possono essere tranquillamente definiti usando le direttive standard di Figura 10; queste direttive, essendo appunto standard, sono compatibili con tutti gli assembler disponibili sul mercato. Le direttive di Figura 13 influiscono solo sull'aspetto estetico dei programmi; molti programmatori preferiscono le direttive di Figura 10 perche' rispecchiano meglio la filosofia della programmazione **Assembly** che non lascia molto spazio alle formalita'.

12.2.3 La direttiva TYPEDEF

Gli assembler piu' recenti mettono a disposizione anche la direttiva **TYPEDEF** che permette di assegnare nuovi nomi (**alias**) alle direttive di Figura 13; in questo modo, e' possibile definire i dati statici di un programma, con una sintassi molto simile (formalmente) a quella utilizzata con i linguaggi di alto livello. I fanatici del **Borland Turbo Pascal** ad esempio, possono scrivere:

```
Integer TYPEDEF SWORD
```

creando in questo modo un alias **Integer** per la direttiva **SWORD**; a questo punto e' possibile scrivere definizioni del tipo:

```
pascalVar Integer +12538
```

12.2.4 Base predefinita per i valori immediati

In assenza di diverse indicazioni da parte del programmatore, tutti i numeri espliciti (valori immediati, spiazzamenti, etc) presenti in un programma **Assembly**, si intendono espressi in base **10**; cio' significa che nella seguente definizione:

```
Pressione DW 3800
```

alla variabile **Pressione** viene assegnato il valore iniziale **3800** espresso in base **10**; se vogliamo indicare in modo esplicito la base numerica di un valore immediato, dobbiamo servirci dei suffissi mostrati in Figura 14.

Figura 14 - Suffissi per la base numerica

Suffisso	Significato	Base
B	Binario	2
O,Q	Ottale	8
D	Decimale	10
H	Esadecimale	16

I suffissi possono essere espressi con una lettera maiuscola o minuscola; per la base ottale e' preferibile utilizzare il suffisso **Q** in quanto la lettera **O** puo' essere confusa con uno zero.
Con questi suffissi, la precedente definizione puo' essere riscritta come:

```
Pressione DW 0000111011011000b
```
oppure:
```
Pressione DW 7330q
```
oppure:
```
Pressione DW 3800d
```
oppure:
```
Pressione DW 0ED8h
```

E' anche possibile imporre una diversa base predefinita attraverso la direttiva:
```
.RADIX n
```
Il valore **n** indica la base predefinita che puo' essere **2**, **8**, **10** o **16**.
Scrivendo ad esempio all'inizio di un programma:
```
.RADIX 16
```
allora tutti i numeri espliciti privi di suffisso si intendono espressi in base **16**; di conseguenza, la precedente definizione:
```
Pressione DW 3800
```
verrebbe interpretata da **MASM** e **TASM** come:
```
Pressione DW 3800h
```
Se si impone una base predefinita **16**, allora tutti i numeri espliciti espressi in una base diversa da **16** (compresi i numeri in base **10**) devono specificare obbligatoriamente il suffisso; come si puo' facilmente intuire, questa situazione puo' creare parecchia confusione, per cui si raccomanda vivamente di lasciare **10** come base predefinita.

Nel caso dei numeri espliciti espressi in base **16**, si puo' presentare un piccolo problema; consideriamo a tale proposito la seguente definizione:
```
VarHex DW F28Ch
```
Quando l'assembler incontra questa definizione, genera un messaggio di errore; il perche' di questo messaggio di errore e' abbastanza evidente. Il valore immediato **F28Ch** inizia con una lettera dell'alfabeto, per cui l'assembler confonde questo valore con un identificatore; se l'identificatore **F28Ch** non esiste, l'assembler genera appunto un messaggio di errore.
Per evitare questo problema, dobbiamo mettere uno zero prima della cifra piu' significativa di **F28Ch**; la precedente definizione diventa quindi:
```
VarHex DW 0F28Ch
```
Nota importante.
Se si utilizzano le direttive **TBYTE** o **DT** (**Define Tenbyte**) per definire una variabile intera, e non si specifica un suffisso per il valore inizializzante, gli assembler come **MASM** e **TASM** si servono del suffisso predefinito **h**; in questo modo, alla variabile viene assegnato un valore intero decimale codificato nel formato **Packed BCD** (che verra' analizzato in un altro capitolo).
In sostanza, se scriviamo ad esempio:
```
bcd_number dt 00386594216072468135
```
alla variabile **bcd_number** viene assegnato il valore esadecimale **00386594216072468135h**!

12.3 Definizione dei dati statici complessi dell'Assembly

Dopo aver parlato dei formati elementari dei dati, che l'**Assembly** ci mette a disposizione, passiamo ad esaminare le direttive che ci permettono di definire i cosiddetti "**aggregati**" di dati; gli aggregati sono particolari strutture dati che possono essere trattate come un singolo dato complesso.
Cominciamo con le strutture propriamente dette, che possono essere create attraverso le direttive **STRUC** e **UNION** disponibili con **MASM** e **TASM**; si tenga presente che altri assembler (e le versioni piu' vecchie di **MASM** e **TASM**) non supportano queste direttive.

12.3.1 La direttiva STRUC

La direttiva **STRUC** permette di creare strutture dati equivalenti alle **struct** del **C/C++** e ai **record** del **Pascal**; con le versioni piu' recenti di **MASM** e **TASM** si puo' usare anche la direttiva **STRUCT**. Una struttura racchiude un insieme di dati di qualsiasi formato; la Figura 15 illustra la sintassi da utilizzare con questa direttiva.

Figura 15 - Direttiva STRUC

```
Struc1 STRUC

   varSt1   dw        ?
   varSt2   dd        ?
   varSt3   dw        ?
   varSt4   db        ?
   varSt5   db        ?

Struc1 ENDS
```

Come si puo' notare, una **STRUC** viene aperta da un nome simbolico seguito dalla direttiva **STRUC**; la struttura viene poi chiusa dallo stesso nome simbolico seguito dalla direttiva **ENDS**, proprio come accade per i segmenti di programma. In effetti, osservando la Figura 15 si puo' constatare che una **STRUC** e' formalmente identica ad un segmento dati; al suo interno infatti, possiamo inserire dati di qualsiasi formato (comprese le strutture stesse).
Nota importante.
Un aspetto importantissimo da sottolineare, e' dato dal fatto che la direttiva **STRUC** non ha lo scopo di definire una struttura dati, ma solo quello di indicare all'assembler le caratteristiche generali della struttura stessa; la dimostrazione di questa affermazione e' data dal fatto che, scrivendo due programmi **Assembly** identici, e inserendo in uno solo di essi la struttura di Figura 15, verranno generati due file eseguibili aventi la stessa dimensione in byte. Cio' indica in modo inequivocabile che l'assembler, incontrando la struttura di Figura 15, non riserva ad essa neanche un byte di memoria; in sostanza, la struttura di Figura 15 rappresenta una cosiddetta **dichiarazione** e non una **definizione**.
Per enfatizzare il fatto che stiamo dichiarando e non definendo una struttura dati, la **STRUC** di Figura 15 deve essere collocata al di fuori di qualsiasi segmento di programma; generalmente, il posto piu' adatto per le varie dichiarazioni e' la parte iniziale di un modulo **Assembly**.

Attraverso le dichiarazioni, stiamo creando in pratica nuovi formati di dati; i nomi attribuiti a questi nuovi formati di dati, possono essere usati come vere e proprie direttive. Nel caso di Figura 15, possiamo utilizzare il nome **Struc1** come se fosse una delle direttive di Figura 10 o di Figura 13; di conseguenza, all'interno di un segmento di programma, possiamo scrivere ad esempio:

```
varStruc1 Struc1 < 2AB1h, 3BF1854Dh, 884Ch, 2Fh, 5Ah >
```

In questo modo stiamo definendo ed inizializzando una struttura **varStruc1** di tipo **Struc1**; come si puo' notare, la lista degli inizializzatori e' racchiusa da una coppia di parentesi angolari. Ricorrendo alla terminologia dei linguaggi di programmazione ad oggetti, si puo' anche dire che **varStruc1** e' una **istanza** di **Struc1**.
E' importante sottolineare ancora la differenza fondamentale che esiste tra una **dichiarazione** e una **definizione**; con la **dichiarazione** di Figura 15 stiamo semplicemente illustrando all'assembler le caratteristiche generali di un dato strutturato come **Struc1**. Attraverso invece la **definizione** di **varStruc1**, stiamo chiedendo all'assembler di riservare fisicamente una locazione di memoria di dimensioni sufficienti a contenere la stessa struttura **varStruc1**; quando l'assembler incontra questa definizione, crea in quel preciso punto del segmento di programma, una locazione di memoria da:

`2 + 4 + 2 + 1 + 1 = 10` byte.

Supponendo che **varStruc1** si trovi all'offset **0028h** di un segmento di programma, allora questa struttura verra' disposta in memoria secondo lo schema di Figura 16.

Figura 16 - varStruc1 in memoria

Membro	varSt5	varSt4	varSt3		varSt2				varSt1	
Contenuto	5Ah	2Fh	88h	4Ch	3Bh	F1h	85h	4Dh	2Ah	B1h
Offset	0031h	0030h	002Fh	002Eh	002Dh	002Ch	002Bh	002Ah	0029h	0028h

Come accade per il linguaggio **C**, anche in **Assembly** gli elementi che fanno parte di una struttura vengono chiamati **membri** (nel **Pascal** si utilizza il termine **campi**); nel nostro caso, la struttura **varStruc1** contiene i membri elencati in Figura 15 o in Figura 16.
Per accedere ai membri di una struttura, si utilizza la stessa sintassi dei linguaggi di alto livello; questa sintassi prevede l'uso dell'operatore **'.'** (punto) secondo la forma:

```
NomeStruttura.NomeMembro
```

NomeStruttura.NomeMembro rappresenta il contenuto della locazione di memoria che si trova all'offset risultante dalla somma:

```
Offset(NomeStruttura) + Offset(NomeMembro)
```

Mentre pero' **Offset(NomeStruttura)** viene calcolato rispetto al segmento di appartenenza della struttura, **Offset(NomeMembro)** viene invece calcolato rispetto ad **Offset(NomeStruttura)**; in sostanza, e' come se **NomeMembro** fosse un dato definito all'interno di un segmento chiamato **NomeStruttura**.
Osservando ad esempio la Figura 15 e la Figura 16, si vede subito che:

varSt1 si trova all'offset **0000h** di **varStruc1**;
varSt2 si trova all'offset **0002h** di **varStruc1**;
varSt3 si trova all'offset **0006h** di **varStruc1**;
varSt4 si trova all'offset **0008h** di **varStruc1**;
varSt5 si trova all'offset **0009h** di **varStruc1**.

Consideriamo ad esempio l'istruzione:

```
MOV AX, DS:varStruc1.varSt1
```

Osservando la Figura 16, si vede subito che l'operando sorgente di questa istruzione fa' riferimento alla **WORD** che si trova all'offset:

```
0028h + 0000h = 0028h
```

del segmento di programma che contiene **varStruc1**; dopo il trasferimento dati si ottiene quindi **AX=2AB1h**.

Consideriamo l'istruzione:

```
MOV EBX, DS:varStruc1.varSt2
```

Osservando la Figura 16, si vede subito che l'operando sorgente di questa istruzione fa' riferimento alla **DWORD** che si trova all'offset:

```
0028h + 0002h = 002Ah
```

del segmento di programma che contiene **varStruc1**; dopo il trasferimento dati si ottiene quindi **EBX=3BF1854Dh**.

Consideriamo l'istruzione:

```
MOV DX, WORD PTR DS:varStruc1.varSt2[2]
```

Osservando la Figura 16, si vede subito che l'operando sorgente di questa istruzione fa' riferimento alla **WORD** che si trova all'offset:

```
0028h + 0002h + 0002h = 002Ch
```

del segmento di programma che contiene **varStruc1**; dopo il trasferimento dati si ottiene quindi **DX=3BF1h** (**WORD** piu' significativa di **varSt2**). In questo caso, l'operatore **WORD PTR** e' necessario in quanto **varSt2** e' stata dichiarata di tipo **DWORD**.

Naturalmente, possiamo accedere a **varStruc1** anche attraverso i registri puntatori; ponendo allora **BX=0028h**, **SI=0002h** e ricordando l'associazione predefinita tra **BX** e **DS**, l'istruzione dell'ultimo esempio puo' essere riscritta come:

```
MOV DX, [BX+SI+2]
```

Sia **MASM** che **TASM** rendono possibile la dichiarazione di strutture che contengono tra i loro membri altre strutture; in un caso del genere si parla di **strutture innestate**.
In relazione alle strutture innestate, bisogna dire che purtroppo tra **MASM** e **TASM** esistono delle differenze che riguardano in particolare la fase di inizializzazione; in Figura 17 vediamo un esempio che mostra una struttura che ha tra i suoi membri un'altra struttura.

Figura 17 - Strutture innestate

```
Point2d STRUC

   x     dw    ?
   y     dw    ?

Point2d ENDS

Rect STRUC

   p1    Point2d  < ?, ? >
   p2    Point2d  < ?, ? >

Rect ENDS
```

In questo esempio, dichiariamo innanzi tutto una struttura **Point2d** che e' formata da due membri, **x** e **y**, entrambi di tipo **WORD**; questi due membri rappresentano le coordinate (ascissa e ordinata) di un punto del piano.
Successivamente dichiariamo una struttura **Rect** che e' formata da due membri, **p1** e **p2**, entrambi di tipo **Point2d**; i due punti **p1** e **p2** rappresentano i vertici, in alto a sinistra e in basso a destra, di un rettangolo.

Se ora vogliamo creare e inizializzare un'istanza di **Rect**, dobbiamo comportarci diversamente a seconda dell'assembler che stiamo usando; nel caso di **MASM** e' possibile scrivere:

```
r1 Rect < < 3AB2h, 1C8Fh >, < 284Dh, 6FC5h > >
```

Come si puo' notare, abbiamo una coppia piu' esterna di parentesi angolari (per **Rect**), che incorpora due coppie interne di parentesi angolari (per i due **Point2d**).

Supponendo che **r1** si trovi all'offset **00F6h** di un segmento di programma, allora questa struttura verra' disposta in memoria secondo lo schema di Figura 18.

Figura 18 - r1 in memoria								
Membro	**p2**				**p1**			
	y		**x**		**y**		**x**	
Contenuto	6Fh	C5h	28h	4Dh	1Ch	8Fh	3Ah	B2h
Offset	00FDh	00FCh	00FBh	00FAh	00F9h	00F8h	00F7h	00F6h

Nel caso di **TASM**, e' possibile ricorrere alle parentesi angolari solo se la struttura da inizializzare non contiene altre strutture innestate; nel caso di Figura 17, ci si deve limitare a scrivere:

```
r1 Rect < ? >
```

spostando la fase di inizializzazione nel blocco codice del programma. La forma usata dal **TASM** per l'inizializzazione di **r1** non e' pero' permessa con il **MASM**; tutto cio' rappresenta un problema nel momento in cui abbiamo la necessita' di scrivere programmi **Assembly** compatibili tra **MASM** e **TASM**.
Fortunatamente, esiste un modo per superare questo ostacolo, e consiste nel creare le istanze di **Rect** utilizzando la seguente sintassi:

```
r1 Rect < >
```

Questa sintassi e' lecita, sia con il **MASM**, sia con il **TASM**, e indica all'assembler di utilizzare per l'inizializzazione di **r1**, gli stessi valori usati nella dichiarazione di **Rect** e **Point2d**; nel nostro caso tutti gli inizializzatori valgono **?**.
A questo punto, nel blocco codice del programma possiamo procedere alla inizializzazione di **r1**; anche nel caso delle strutture innestate, l'accesso ai membri avviene con la stessa sintassi dei linguaggi di alto livello. Questa sintassi assume la forma:

```
NomeStruttura.NomeStrutturaInnestata.NomeMembro
```

Come al solito, **NomeStruttura.NomeStrutturaInnestata.NomeMembro** rappresenta il contenuto della locazione di memoria che si trova all'offset risultante dalla somma:

```
Offset(NomeStruttura) + Offset(NomeStrutturaInnestata) + Offset(NomeMembro)
```

Offset(NomeStruttura) viene calcolato rispetto al segmento di appartenenza della struttura;
Offset(NomeStrutturaInnestata) viene calcolato rispetto a **Offset(NomeStruttura)**;
Offset(NomeMembro) viene calcolato rispetto a **Offset(NomeStrutturaInnestata)**.

Osservando che **r1.p1** rappresenta una struttura **Point2d** che contiene il vertice in alto a sinistra del rettangolo, allora **r1.p1.x** e **r1.p1.y** rappresentano le coordinate di questo stesso vertice; l'offset di **p1** viene calcolato rispetto a **r1**, mentre gli offset di **x** e di **y** vengono calcolati rispetto a **p1**.
Analogamente, osservando che **r1.p2** rappresenta una struttura **Point2d** che contiene il vertice in basso a destra del rettangolo, allora **r1.p2.x** e **r1.p2.y** rappresentano le coordinate di questo stesso vertice; l'offset di **p2** viene calcolato rispetto a **r1**, mentre gli offset di **x** e di **y** vengono calcolati rispetto a **p2**.

Se allora vogliamo inizializzare la struttura **r1** nel blocco codice del programma, possiamo scrivere le istruzioni mostrate in Figura 19; queste istruzioni rappresentano chiaramente dei trasferimenti di dati da **Imm16** a **Mem16**.

Figura 19 - Inizializzazione di r1

```
MOV      DS:r1.p1.x, 3AB2h
MOV      DS:r1.p1.y, 1C8Fh
MOV      DS:r1.p2.x, 284Dh
MOV      DS:r1.p2.y, 6FC5h
```

Osservando lo schema di Figura 18, si puo' constatare che:

r1.p1.x rappresenta il contenuto della locazione di memoria da **16** bit che si trova all'offset:
`00F6 + 0000h + 0000h = 00F6`
r1.p1.y rappresenta il contenuto della locazione di memoria da **16** bit che si trova all'offset:
`00F6 + 0000h + 0002h = 00F8`
r1.p2.x rappresenta il contenuto della locazione di memoria da **16** bit che si trova all'offset:
`00F6 + 0004h + 0000h = 00FA`
r1.p2.y rappresenta il contenuto della locazione di memoria da **16** bit che si trova all'offset:
`00F6 + 0004h + 0002h = 00FC`

Una struttura innestata puo' avere tra i suoi membri ulteriori strutture innestate; gli innesti possono andare avanti sino all'esaurimento della memoria disponibile.

Un'altra notevole differenza tra **TASM** e **MASM** e' data dal fatto che per **MASM**, i nomi dei membri di una struttura sono invisibili all'esterno della struttura stessa, e quindi possono essere ridefiniti (ad esempio come membri di altre strutture); in sostanza, nel caso del **MASM** l'identificatore **r1.p1.x** viene considerato distinto da **p1** o da **x**. Per il **TASM** invece, i nomi dei membri di una struttura hanno visibilita' globale (estesa a tutto il modulo di appartenenza), e quindi non possono essere ridefiniti nello stesso modulo; per maggiori dettagli, si consiglia di fare riferimento alla documentazione dell'assembler che si sta utilizzando.

12.3.2 La direttiva UNION

Passiamo ora alla direttiva **UNION** (unione) attraverso la quale possiamo creare aggregati di dati, molto simili alle strutture; le **UNION** dell'**Assembly** equivalgono alle **union** del **C/C++** e ai **record varianti** del **Pascal**.
La differenza fondamentale che esiste tra una **STRUC** e una **UNION** sta nel fatto che tutti i membri di una **UNION** vengono sovrapposti tra loro in modo che condividano lo stesso offset iniziale in memoria; questo tipo di aggregato quindi e' molto utile quando si ha bisogno di una variabile che in fase di esecuzione di un programma, assume formati diversi (**BYTE**, **WORD**, etc).
Anche nel caso delle **UNION** esistono delle incompatibilita' tra **MASM** e **TASM**, comprese quelle gia' elencate per le **STRUC**; per evitare questo problema, la cosa migliore da fare consiste nell'effettuare le varie inizializzazioni nel blocco codice del programma.
Come si puo' facilmente intuire, nel caso delle **UNION** possiamo inizializzare solo un membro per volta; la Figura 20 mostra un esempio di dichiarazione di una unione.

Figura 20 - Direttiva UNION

```
Union1 UNION

   varUn1   db        ?
   varUn2   dw        ?
   varUn3   dd        ?

Union1 ENDS
```

Osserviamo che **Union1** dichiara una **UNION** formata da un membro di tipo **BYTE**, uno di tipo **WORD** e uno di tipo **DWORD**; se ora definiamo in un segmento di programma una istanza **VarUnion1** di **Union1**, l'assembler riserva a questa istanza uno spazio la cui ampiezza in bit e' pari a quella del membro piu' grande di **Union1**. Come possiamo notare in Figura 20, il membro piu' grande di **Union1** e' **varUn3** che ha una ampiezza di **32** bit; lo spazio da **32** bit riservato dall'assembler e' sufficiente a contenere uno qualunque dei tre membri di Figura 20.
Per la definizione di **varUnion1** utilizziamo la forma compatibile con **MASM** e **TASM**, e cioe':

```
varUnion1 Union1 < >
```

In questo caso stiamo dicendo all'assembler che per l'inizializzazione di **varUnion1** utilizziamo i valori predefiniti di Figura 20; a questo punto, nel blocco codice del programma possiamo scrivere ad esempio:

```
MOV DS:varUnion1.varUn2, 2BFFh
```

Abbiamo quindi inizializzato il membro **varUn2** di tipo **WORD**; supponendo che **varUnion1** si trovi all'offset **00D4h** di un segmento di programma, allora questa **UNION** assumera' in memoria lo schema illustrato in Figura 21.

<table>
<tr><th colspan="5">Figura 21 - varUnion1 in memoria</th></tr>
<tr><th rowspan="3">Membro</th><td colspan="3">_</td><th>varUn1</th></tr>
<tr><td colspan="2">_</td><th colspan="2">varUn2</th></tr>
<tr><th colspan="4">varUn3</th></tr>
<tr><th>Contenuto</th><td>?</td><td>?</td><td>2Bh</td><td>FFh</td></tr>
<tr><th>Offset</th><td>00D7h</td><td>00D6h</td><td>00D5h</td><td>00D4h</td></tr>
</table>

Osserviamo subito che i tre membri di **varUnion1** condividono tutti lo stesso offset iniziale **00D4h**; siccome abbiamo inizializzato il membro **varUn2** a **16** bit, solamente i primi **16** bit di **varUnion1** sono significativi. In sostanza, subito dopo l'inizializzazione, solamente il membro **varUnion1.varUn2** contiene un dato valido; durante la fase di esecuzione del programma, possiamo alterare in qualunque momento questa situazione scrivendo ad esempio:

```
MOV DS:varUnion1.varUn1, 2Fh
```

Subito dopo l'esecuzione di questa istruzione, la locazione di Figura 21 assume l'aspetto mostrato in Figura 22.

<table>
<tr><th colspan="5">Figura 22 - varUnion1 in memoria</th></tr>
<tr><th rowspan="3">Membro</th><td colspan="3">_</td><th>varUn1</th></tr>
<tr><td colspan="2">_</td><th colspan="2">varUn2</th></tr>
<tr><th colspan="4">varUn3</th></tr>
<tr><th>Contenuto</th><td>?</td><td>?</td><td>?</td><td>2Fh</td></tr>
<tr><th>Offset</th><td>00D7h</td><td>00D6h</td><td>00D5h</td><td>00D4h</td></tr>
</table>

Da questo momento, solamente **varUnion1.varUn1** contiene un dato valido; questa situazione permane sino alla prossima modifica di **varUnion1**.
Possiamo dire quindi che, durante la fase di esecuzione del programma, **varUnion1** puo' essere utilizzata per contenere a scelta, un dato a **8** bit, oppure un dato a **16** bit, oppure un dato a **32** bit; chiaramente, e' compito del programmatore tenere traccia del membro che e' valido in un determinato momento.

Una **UNION** puo' contenere tra i suoi membri anche una o piu' **STRUC**; a sua volta, ogni **STRUC** innestata puo' contenere altre strutture innestate, o anche altre unioni innestate. La gestione di questi innesti e' tanto piu' complessa quanto piu' sono "contorti" gli innesti stessi; in ogni caso, raramente si ha bisogno di strutture dati cosi' complicate.

La Figura 23 illustra un esempio di **UNION** che contiene tra i suoi membri anche una **STRUC**; anche in questo caso ricorriamo alla inizializzazione compatibile con **MASM** e **TASM**.

Figura 23 - UNION con STRUC innestata

```
Agrumi   STRUC

   Arancio  dd    ?
   Limone   dd    ?

Agrumi   ENDS

Frutta   UNION

   Mela     db       ?
   Pera     dw       ?
   Ciliegia dd       ?
   Altro    Agrumi   < ?, ? >

Frutta   ENDS
```

La **UNION** chiamata **Frutta** puo' contenere un **BYTE** di nome **Mela**, oppure una **WORD** di nome **Pera**, oppure una **DWORD** di nome **Ciliegia**, oppure una struttura **Agrumi** di nome **Altro**; la struttura **Agrumi** a sua volta e' formata da due **DWORD** e occupa quindi **64** bit.
In seguito a questa dichiarazione, possiamo utilizzare il nome **Frutta** per definire le istanze di questa **UNION**; nel segmento dati del nostro programma possiamo scrivere ad esempio:

```
varFrutta1 Frutta < >
```

Quando l'assembler incontra questa definizione, riserva a **varFrutta1** uno spazio sufficiente a contenere il membro piu' grande dell'unione **Frutta**; dalla Figura 23 si rileva che il membro piu' grande e' **Altro** che occupa **64** bit.
Nel segmento di codice del nostro programma possiamo ora procedere con l'inizializzazione di uno dei membri di **varFrutta1**; supponendo di voler inizializzare per primo il membro **Pera** a **16** bit, possiamo scrivere ad esempio:

```
MOV DS:varFrutta1.Pera, 8B21h
```

Nell'ipotesi che **varFrutta1** si trovi all'offset **00C8h** di un segmento di programma, allora questa **UNION** assumera' in memoria lo schema illustrato in Figura 24.

Figura 24 - varFrutta1 in memoria

Membro	_							**Mela**
	_						**Pera**	
	_				**Ciliegia**			
	Limone				**Arancio**			
Contenuto	?	?	?	?	?	?	8Bh	21h
Offset	00CFh	00CEh	00CDh	00CCh	00CBh	00CAh	00C9h	00C8h

Da questo momento, solamente **varFrutta1.Pera** contiene un dato valido; questa situazione permane sino alla prossima modifica di **varFrutta1**. Ad un certo punto della fase di esecuzione, possiamo decidere di alterare la situazione di Figura 24, inizializzando **varFrutta1.Altro**; possiamo scrivere ad esempio:

```
MOV DS:varFrutta1.Altro.Arancio, 3FAB819Ch
```

e:

```
MOV DS:varFrutta1.Altro.Limone, 6DF934E1h
```

Dopo l'esecuzione di queste istruzioni, la locazione di memoria di Figura 24 assume l'aspetto mostrato in Figura 25.

<table>
<tr><th colspan="9">Figura 25 - varFrutta1 in memoria</th></tr>
<tr><th rowspan="4">Membro</th><td colspan="7">_</td><th>Mela</th></tr>
<tr><td colspan="6">_</td><th colspan="2">Pera</th></tr>
<tr><td colspan="4">_</td><th colspan="4">Ciliegia</th></tr>
<tr><th colspan="4">Limone</th><th colspan="4">Arancio</th></tr>
<tr><th>Contenuto</th><td>6Dh</td><td>F9h</td><td>34h</td><td>E1h</td><td>3Fh</td><td>ABh</td><td>81h</td><td>9Ch</td></tr>
<tr><th>Offset</th><td>00CFh</td><td>00CEh</td><td>00CDh</td><td>00CCh</td><td>00CBh</td><td>00CAh</td><td>00C9h</td><td>00C8h</td></tr>
</table>

Da questo momento, solamente **varFrutta1.Altro** contiene un dato valido; questa situazione permane sino alla prossima modifica di **varFrutta1**.

Tutte le considerazioni appena illustrate per le **UNION**, sono valide anche per le **STRUC**; dagli esempi che sono stati presentati si puo' anche constatare che seguendo la logica, la gestione di questi aggregati complessi non presenta grosse difficolta'. Nei casi piu' contorti, possiamo semplificarci notevolmente la vita tracciando su un foglio di carta uno schema della locazione di memoria che contiene l'aggregato che vogliamo gestire; gli schemi come quelli di Figura 22 o di Figura 25, ci permettono anche di capire meglio il modo di lavorare della **CPU**.

12.3.3 La direttiva RECORD

Attraverso la direttiva **RECORD** possiamo dichiarare una struttura contenente una sequenza di dati, ciascuno dei quali puo' avere una differente ampiezza in bit; questa direttiva si rivela molto utile nel momento in cui abbiamo la necessita' di compattare nel piu' piccolo spazio possibile, una numerosa serie di informazioni. Un esempio pratico e' rappresentato dal registro **FLAGS** della **CPU**, dove ogni singolo bit ha un preciso significato; il registro **FLAGS** puo' essere visto quindi come un classico esempio di **RECORD**.

La sintassi da utilizzare per la dichiarazione di un **RECORD** e' la seguente:

```
NomeRecord RECORD NomeMembro1: ampiezza1, NomeMembro2: ampiezza2, ...
```

Consideriamo ad esempio la seguente dichiarazione:

```
RecData RECORD Giorno: 5, Mese: 4, Anno: 11
```

Il record **RecData** memorizza in forma compatta una data del calendario; sono presenti tre membri destinati a memorizzare il giorno, il mese e l'anno corrente. Osserviamo che:

***** il membro **Giorno** e' formato da **5** bit e puo' quindi contenere un valore compreso tra **0** e **31**;
***** il membro **Mese** e' formato da **4** bit e puo' quindi contenere un valore compreso tra **0** e **15**;
***** il membro **Anno** e' formato da **11** bit e puo' quindi contenere un valore compreso tra **0** e **2047**.

Attraverso la dichiarazione **RecData** possiamo ora definire istanze di questo **RECORD**; la sintassi da utilizzare e' la stessa delle strutture. Supponiamo ad esempio di voler creare l'istanza **varData1** che codifica la data **24/10/2003** (in binario **11000b/1010b/11111010011b**); in un segmento del nostro programma possiamo scrivere allora:

```
varData1 RecData < 24, 10, 2003 >
```

Quando l'assembler incontra questa definizione, riserva uno spazio sufficiente a contenere l'intero **RECORD**; nel caso delle **CPU 80286** e inferiori, l'ampiezza di un **RECORD** puo' essere di **8** o **16** bit. Con le **CPU 80386** e superiori si puo' avere anche una ampiezza di **32** bit; l'assembler, eventualmente, incrementa la dimensione complessiva del **RECORD**, in modo da portarla a **8** bit, **16** bit o **32** bit. Questo incremento viene ottenuto inserendo un numero adeguato di zeri alla sinistra del **RECORD** stesso; nel caso di **varData1**, la dimensione complessiva e':
`5 + 4 + 11 = 20` bit.
Siccome questa dimensione e' maggiore di **16** bit, l'assembler crea una locazione di memoria da **32** bit aggiungendo **12** zeri alla sinistra del **RECORD**; naturalmente, in questo caso dobbiamo disporre almeno di una **CPU 80386**.

Nota importante.
Bisogna prestare particolare attenzione al fatto che, a differenza di quanto accade con le **STRUC**, i vari membri di un **RECORD** formano un unico numero binario; nel caso quindi del nostro esempio, **Anno** rappresenta la parte meno significativa di **varData1**, mentre **Giorno** rappresenta la parte piu' significativa. La definizione di **varData1** crea quindi una locazione di memoria contenente il numero binario **11000101011111010011b**; in pratica, e' come se il programmatore avesse scritto:

```
varData1 DD 00000000000011000101011111010011b
```

Nell'ipotesi che **varData1** si trovi in memoria all'offset **008Dh** di un segmento di programma, si ottiene la situazione illustrata in Figura 26.

<table>
<tr><th colspan="33">Figura 26 - varData1 in memoria</th></tr>
<tr><th>Membro</th><th colspan="12">_</th><th colspan="5">Giorno</th><th colspan="4">Mese</th><th colspan="11">Anno</th></tr>
<tr><th>Contenuto</th><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><th>Offset</th><td colspan="8">0090h</td><td colspan="8">008Fh</td><td colspan="8">008Eh</td><td colspan="8">008Dh</td></tr>
</table>

Osserviamo ancora una volta che **Anno** occupa gli **11** bit meno significativi di **varData1**, mentre **Giorno** occupa i **5** bit piu' significativi; subito dopo **Giorno** troviamo inoltre i **12** zeri aggiunti dall'assembler.

In teoria, per accedere ai membri di un **RECORD** si puo' utilizzare la stessa sintassi gia' vista per le strutture; le versioni piu' recenti di **MASM** e **TASM** non permettono pero' questa possibilita'. Del resto, osservando la Figura 26 si puo' facilmente constatare che non e' possibile maneggiare i membri di un **RECORD** come si fa' con le strutture; il programmatore deve quindi trattare un **RECORD** come se fosse un singolo numero binario dove ogni gruppo di bit ha un preciso significato. Nei capitoli successivi verranno illustrate numerose istruzioni che ci permettono di accedere in lettura e in scrittura a uno o piu' bit di una locazione di memoria.

12.3.4 La direttiva DUP

Attraverso la direttiva **DUP** e' possibile "replicare" un oggetto che puo' essere, un dato semplice, un aggregato di dati, o persino un'altra direttiva **DUP** seguita da un ulteriore oggetto da replicare; a differenza di quanto accade con **STRUC**, **UNION** e **RECORD** che vengono usate nelle dichiarazioni, la direttiva **DUP** deve essere inserita in un segmento di programma in quanto comporta la definizione di un aggregato di dati, con conseguente allocazione della memoria.

La sintassi generale per **DUP** e' la seguente:

```
NomeSimbolico DIRETTIVA Repliche DUP ( Oggetto )
```

La **DIRETTIVA** e' una di quelle illustrate in Figura 10 e in Figura 13, oppure il nome simbolico di una **STRUC**, **UNION**, **RECORD**, etc; il valore immediato **Repliche** indica il numero di oggetti da replicare.
Gli oggetti replicati con **DUP** vengono disposti in memoria in modo consecutivo e contiguo; indicando con **Dimensione(Oggetto)** la dimensione in byte di **Oggetto**, possiamo dire allora che la memoria complessiva da allocare e' pari al prodotto:

```
Repliche * Dimensione(Oggetto)
```

Consideriamo il seguente esempio:

```
VettWord1 DW 4 DUP ( 03FDh )
```

Quando l'assembler incontra questa definizione, riserva uno spazio pari a **4** word, per un totale di **4*2=8** byte; in questo spazio vengono sistemate **4 WORD**, ciascuna delle quali viene inizializzata con il valore **03FDh**.
Nell'ipotesi che **VettWord1** si trovi all'offset **00F2h** di un segmento di programma, si ottiene per questa variabile la disposizione in memoria mostrata in Figura 27.

Figura 27 - VettWord1 in memoria

Indice	3		2		1		0	
Contenuto	03h	FDh	03h	FDh	03h	FDh	03h	FDh
Offset	00F9h	00F8h	00F7h	00F6h	00F5h	00F4h	00F3h	00F2h

Dalla Figura 27 si rileva chiaramente che:

***** La **WORD** di indice **0** e' individuata da **VettWord1[0]**;
***** La **WORD** di indice **1** e' individuata da **VettWord1[2]**;
***** La **WORD** di indice **2** e' individuata da **VettWord1[4]**;
***** La **WORD** di indice **3** e' individuata da **VettWord1[6]**.

Caricando in **BX** l'offset **00F2h** e ricordando l'associazione predefinita tra **BX** e **DS**, possiamo anche dire che:

***** La **WORD** di indice **0** e' individuata da **[BX+0]**;
***** La **WORD** di indice **1** e' individuata da **[BX+2]**;
***** La **WORD** di indice **2** e' individuata da **[BX+4]**;
***** La **WORD** di indice **3** e' individuata da **[BX+6]**.

Una sequenza consecutiva e contigua di oggetti, tutti della stessa natura, viene chiamata **vettore**; ogni oggetto appartenente ad un vettore prende il nome di **elemento**.
L'esempio appena illustrato ci permette di sottolineare una importante differenza che esiste tra l'**Assembly** e i linguaggi di alto livello nella gestione dei vettori; nei linguaggi di alto livello, i **4**

elementi di Figura 27 vengono individuati dai nomi:
VettWord1[0], **VettWord1[1]**, **VettWord1[2]**, **VettWord1[3]**.
Il calcolo degli spiazzamenti viene affidato in questo caso al compilatore o all'interprete; nel caso ad esempio di **VettWord1[3]**, l'indice **3** viene moltiplicato per **2** (dimensione in byte di ogni elemento del vettore), in modo da ottenere **VettWord1[6]**.
In **Assembly** invece, tutti questi calcoli spettano al programmatore; del resto, l'**Assembly** viene utilizzato proprio da chi vuole avere il controllo totale sul programma che sta scrivendo!

Passiamo ora ad un esempio piu' impegnativo; in riferimento alla struttura **Rect** dichiarata in Figura 17, consideriamo la seguente definizione:

```
VettRect1 Rect 10 DUP ( < > )
```

Davanti a questa definizione, l'assembler non si spaventa per niente e crea uno spazio sufficiente per contenere **10** strutture **Rect**; ogni **Rect** occupa **8** byte, per cui la memoria totale allocata dall'assembler e' pari a **10*8=80** byte.
Supponendo che **VettRect1** si trovi all'offset **00F6h** di un segmento di programma, allora questo vettore di oggetti **Rect** verra' disposto in memoria secondo lo schema di Figura 28; in questa figura, per ovvie ragioni vengono mostrati solamente i primi due **Rect** del vettore.

Figura 28 - VettRect1 in memoria

Indice	1								0							
Membro	**p2**				**p1**				**p2**				**p1**			
	y		**x**		**y**		**x**		**y**		**x**		**y**		**x**	
Contenuto	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
Offset	0105h	0104h	0103h	0102h	0101h	0100h	00FFh	00FEh	00FDh	00FCh	00FBh	00FAh	00F9h	00F8h	00F7h	00F6h

Analizzando la Figura 28 rileviamo che il **Rect** di indice **0** e' individuato da **VettRect1[0]**; di conseguenza:

VettRect1[0].p1 rappresenta il **Point2d** che si trova all'offset:

```
00F6h + 0000h + 0000h = 00F6h
```

VettRect1[0].p1.x rappresenta la **WORD** che si trova all'offset:

```
00F6h + 0000h + 0000h + 0000h = 00F6h
```

VettRect1[0].p1.y rappresenta la **WORD** che si trova all'offset:

```
00F6h + 0000h + 0000h + 0002h = 00F8h
```

Analogamente:

VettRect1[0].p2 rappresenta il **Point2d** che si trova all'offset:

```
00F6h + 0000h + 0004h = 00FAh
```

VettRect1[0].p2.x rappresenta la **WORD** che si trova all'offset:

```
00F6h + 0000h + 0004h + 0000h = 00FAh
```

VettRect1[0].p2.y rappresenta la **WORD** che si trova all'offset:

```
00F6h + 0000h + 0004h + 0002h = 00FCh
```

Sempre dalla Figura 28 rileviamo che ogni **Rect** occupa **8** byte, per cui il **Rect** di indice **1** e' individuato da **VettRect1[8]**; di conseguenza:

VettRect1[8].p1 rappresenta il **Point2d** che si trova all'offset:
```
00F6h + 0008h + 0000h = 00FEh
```
VettRect1[8].p1.x rappresenta la **WORD** che si trova all'offset:
```
00F6h + 0008h + 0000h + 0000h = 00FEh
```
VettRect1[8].p1.y rappresenta la **WORD** che si trova all'offset:
```
00F6h + 0008h + 0000h + 0002h = 0100h
```

Analogamente:

VettRect1[8].p2 rappresenta il **Point2d** che si trova all'offset:
```
00F6h + 0008h + 0004h = 0102h
```
VettRect1[8].p2.x rappresenta la **WORD** che si trova all'offset:
```
00F6h + 0008h + 0004h + 0000h = 0102h
```
VettRect1[8].p2.y rappresenta la **WORD** che si trova all'offset:
```
00F6h + 0008h + 0004h + 0002h = 0104h
```

Per accedere agli altri **Rect** si utilizza lo stesso meccanismo; infatti, il **Rect** di indice **2** e' individuato da **VettRect1[16]**, il **Rect** di indice **3** e' individuato da **VettRect1[24]** e cosi' via.

Volendo utilizzare i registri puntatori, possiamo porre **BX=00F6h**; a questo punto osserviamo che:

VettRect1[Indice*8].p1.x equivale a **[BX+(Indice*8)+0+0]**;
VettRect1[Indice*8].p1.y equivale a **[BX+(Indice*8)+0+2]**;
VettRect1[Indice*8].p2.x equivale a **[BX+(Indice*8)+4+0]**;
VettRect1[Indice*8].p2.y equivale a **[BX+(Indice*8)+4+2]**.

Volendo trasferire ad esempio in **AX** il contenuto del membro **p2.y** del **Rect** di indice **3**, dobbiamo scrivere l'istruzione:
```
MOV AX, [BX+(3*8)+4+2]
```
L'espressione **(3*8)+4+2** non comporta nessuna perdita di tempo in quanto viene risolta dall'assembler in fase di assemblaggio del programma; alla fine viene generato un codice macchina contenente un semplice **effective address** del tipo **[BX+Disp16]**.

Come e' stato detto in precedenza, l'oggetto replicato da **DUP** puo' essere persino un'altra direttiva **DUP** seguita da un ulteriore oggetto da replicare; possiamo scrivere ad esempio:
```
MatrWord1 DW 5 DUP (8 DUP (72F8h))
```
In questo modo stiamo chiedendo all'assembler di replicare **5** oggetti, ciascuno dei quali e' un vettore di **8 WORD** che valgono tutte **72F8h**; complessivamente abbiamo bisogno quindi di **5*8=40** word, per un totale di **40*2=80** byte.

Un "vettore di vettori" prende il nome di **matrice** o **vettore bidimensionale**; ogni oggetto della matrice prende il nome di **elemento**.

La matrice **MatrWord1** puo' essere schematizzata simbolicamente come in Figura 29.

Figura 29 - Matrice MatrWord1

		Colonna							
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Riga	**0**	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h
	1	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h
	2	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h
	3	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h
	4	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h	72F8h

Naturalmente, **MatrWord1** viene disposta in memoria, non come in Figura 29, ma come una sequenza di **5*8=40 WORD** consecutive e contigue; il programmatore **Assembly** e' libero di gestire come meglio crede questa disposizione. Diversi linguaggi di alto livello dispongono in memoria le matrici, come una sequenza consecutiva e contigua di righe; in questo caso si dice che la matrice e' **row ordered** (ordinata per righe). Altri linguaggi invece, dispongono in memoria le matrici, come una sequenza consecutiva e contigua di colonne; in questo caso si dice che la matrice e' **column ordered** (ordinata per colonne).

Tornando a **MatrWord1**, possiamo osservare che i vari elementi di questa matrice sono individuati dalla sequenza:
MatrWord1[0], **MatrWord1[2]**, **MatrWord1[4]**, ...
Se abbiamo tracciato su un foglio di carta uno schema come quello di Figura 29, si presenta il problema di come risalire all'indirizzo di memoria dell'elemento che si trova all'incrocio tra un certo indice di riga e un certo indice di colonna; in sostanza, dato l'elemento di Figura 29 che si trova all'incrocio tra la riga **i** e la colonna **j**, vogliamo determinare lo spiazzamento del corrispondente elemento che si trova in memoria.
Indicando con **Dimensione(elemento)** la dimensione in byte di ciascun elemento della matrice, la formula da utilizzare e' la seguente:
`Spiazzamento = ((i * numero_colonne) + j) * Dimensione(elemento)`
Naturalmente, lo **Spiazzamento** cosi' calcolato deve essere sommato all'offset da cui inizia **MatrWord1** in memoria; ad esempio, l'elemento che si trova all'incrocio tra la riga **i=3** e la colonna **j=4**, corrisponde a:
`MatrWord1[((3*8)+4)*2]` cioe' `MatrWord1[56]`

La direttiva **DUP** puo' essere utilizzata anche come membro di una **STRUC** o di una **UNION**; la Figura 30 mostra un esempio pratico che fa' riferimento al **RECORD RecData** definito in precedenza.

Figura 30 - Uso di DUP nelle strutture

```
Automobile  STRUC

   CodModello  dw       ?
   Cilindrata  dw       ?
   Cavalli     dw       ?
   Revisioni   RecData  10 DUP ( < > )

Automobile  ENDS
```

Questa struttura e' formata da **3** word e da un vettore di **10 RecData**; ogni **RecData** occupa **32** bit (**4** byte), per cui la dimensione totale di **Automobile** e' pari a:
`2 + 2 + 2 + (10 * 4) = 6 + 40 = 46` byte.
In un segmento di programma possiamo ora creare delle istanze di **Automobile**; la seguente definizione:
```
FerrariF3000 Automobile 20 DUP ( < > )
```
crea un vettore di **20** strutture **Automobile**; questo vettore richiede quindi **46*20=920** byte di memoria.
I vari elementi di questo vettore sono:

FerrariF3000[0]
FerrariF3000[46]
FerrariF3000[92]
FerrariF3000[138]
....

In relazione all'elemento **FerrariF3000[46]**, e tenendo presente che ogni **RecData** occupa **4** byte, possiamo dire che i vari elementi del vettore **Revisioni** sono:

FerrariF3000[46].Revisioni[0]
FerrariF3000[46].Revisioni[4]
FerrariF3000[46].Revisioni[8]
FerrariF3000[46].Revisioni[12]
....

Volendo inserire in **FerrariF3000[46].Revisioni[0]** la data **12/03/2008** (in binario **01100b/0011b/11111011000b**), possiamo scrivere l'istruzione:
```
MOV DS:FerrariF3000[46].Revisioni[0], 01100001111111011000b
```
Osserviamo che ogni elemento **Revisioni[i]** e' una **DWORD**; di conseguenza, l'assembler e' in grado di rilevare che questa istruzione rappresenta un trasferimento dati da **Imm32** a **Mem32**.

Utilizzando numerose direttive **DUP** innestate, si possono ottenere vettori tridimensionali, quadridimensionali, etc; tutto cio' si applica non solo a oggetti semplici, ma anche a oggetti di tipo struttura, unione, etc. In questo modo si ottengono aggregati di dati particolarmente complessi; la dimensione complessiva di ciascuno di questi aggregati non deve superare **65536** byte.

12.3.5 Creazione diretta di vettori monodimensionali e multidimensionali

Tutte le definizioni effettuate con la direttiva **DUP** possono essere ottenute anche in modo diretto; cio' e' possibile in quanto, in fase di definizione di un dato, l'assembler ci permette di specificare una lista formata da uno o piu' inizializzatori separati da virgole.
Ad esempio, la definizione:
```
VettWord1 DW 8 DUP ( 03BCh )
```
equivale a:
```
VettWord1 DW 03BCh, 03BCh, 03BCh, 03BCh, 03BCh, 03BCh, 03BCh, 03BCh
```
Anche in questo caso quindi, l'assembler crea una locazione di memoria da **8** word, nella quale vengono disposte in modo consecutivo e contiguo, **8 WORD** inizializzate tutte con il valore fisso **03BCh**.
Il vantaggio del metodo diretto sta nel fatto che possiamo specificare una lista di inizializzatori, tutti diversi tra loro; possiamo scrivere ad esempio:
```
VettWord1 DW 2800, 3500, 1890, 3961, 8767, 9945, 1998, 6780
```

Se vogliamo simulare una matrice da **4*10 BYTE**, possiamo ricorrere alla direttiva **DUP** scrivendo:

```
MatrByte1 DB 4 DUP ( 10 DUP ( 0 ) )
```

Volendo utilizzare il metodo diretto, possiamo assegnare un valore diverso ad ogni elemento della matrice; in questo modo si ottiene la situazione mostrata in Figura 31 (ricordiamo che i nomi simbolici da assegnare ai dati sono facoltativi).

Figura 31 - Definizione di MatrByte1

```
MatrByte1   db    3, 5, 4, 9, 2, 2, 1, 4, 4, 8
            db    1, 8, 8, 8, 4, 3, 2, 1, 0, 0
            db    4, 4, 2, 2, 7, 5, 3, 8, 9, 9
            db    2, 2, 6, 8, 4, 1, 6, 8, 1, 0
```

Questa matrice e' chiaramente ordinata per righe

Capitolo 13 - Assembling & Linking

Lo scopo di questo capitolo e' quello di illustrare il procedimento che bisogna seguire, per convertire in formato eseguibile un programma scritto in linguaggio **Assembly**; a tale proposito, ci serviremo del modello presentato nel precedente capitolo.

13.1 Programma Assembly di esempio

La Figura 1 illustra un programma di esempio chiamato **EXETEST.ASM**, a partire dal quale otterremo l'eseguibile finale; questo programma fa' riferimento ai vari dati, elementari e complessi, illustrati nel precedente capitolo.

Figura 1 - Programma Assembly di esempio

```
;--------------------------------------------------;
; File exetest.asm                                 ;
; Esempio di programma Assembly in formato EXE     ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                            ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h              ; 1024 byte per lo stack

; dichiarazione record RecData

RecData RECORD Giorno: 5, Mese: 4, Anno: 11

; dichiarazione struttura Rect

Point2d STRUC

   x     dw    ?
   y     dw    ?

Point2d ENDS

Rect STRUC

   p1    Point2d  < ?, ? >
   p2    Point2d  < ?, ? >

Rect ENDS

; dichiarazione struttura Automobile

Automobile  STRUC

   CodModello  dw       ?
   Cilindrata  dw       ?
   Cavalli     dw       ?
   Revisioni   RecData  10 DUP ( < > )

Automobile  ENDS
```

```
;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;----- inizio definizione variabili  statiche -----

varByte     db          0bh                     ; intero a 1 byte
varWord     dw          0abcdh                  ; intero a 2 byte
varDword    dd          12345678h               ; intero a 4 byte
varQword    dq          123456789abcdef0h       ; intero a 6 byte
varTbyte    dt          00001111222233334444h   ; intero a 10 byte
varFloat32  dd          3.14                    ; reale a 4 byte
varFloat64  dq          -5.12345E-200           ; reale a 8 byte

DataViaggio RecData     < 12, 06, 2001 >        ; RecData a 4 byte
varRect     Rect        < >                     ; Rect a 8 byte
vettRect    Rect        10 dup ( < > )          ; vettore di 10 Rect
FiatPanda   Automobile  < >                     ; Automobile a 46 byte
vettAuto    Automobile  20 dup ( < > )          ; vettore di 20 Automobile

;------- fine definizione variabili statiche ------

DATASEGM    ENDS

;############### segmento codice #################

CODESEGM    SEGMENT  DWORD PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; associa CODESEGM a CS

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; associa DATASEGM a DS

;------- inizio blocco principale istruzioni ------

; inizializzazione di varRect

   mov      varRect.p1.x, 3500
   mov      varRect.p1.y, 8000
   mov      varRect.p2.x, 6850
   mov      varRect.p2.y, 9700

; inizializzazione di vettRect[0] (elemento di indice 0)

   mov      vettRect[0].p1.x, 3cf2h
   mov      vettRect[0].p1.y, 8dabh
   mov      vettRect[0].p2.x, 5c2bh
   mov      vettRect[0].p2.y, 66f1h

; inizializzazione di vettRect[8] (elemento di indice 1)

   mov      vettRect[8].p1.x, 93f1h
   mov      vettRect[8].p1.y, 359ch
   mov      vettRect[8].p2.x, 86ffh
   mov      vettRect[8].p2.y, 12bch

; inizializzazione di FiatPanda

   mov      FiatPanda.CodModello, 3518
```

```
   mov      FiatPanda.Cilindrata, 1000
   mov      FiatPanda.Cavalli, 45

; inizializzazione di FiatPanda.Revisioni[0] con la data
; 12/3/2008 = 1100b/00011b/11111011000b

   mov      FiatPanda.Revisioni[0], 1100000111111011000b

; inizializzazione di FiatPanda.Revisioni[4] con la data
; 12/3/2010 = 1100b/00011b/11111011010b

   mov      FiatPanda.Revisioni[4], 1100000111111011010b

; inizializzazione di vettAuto[0] (elemento di indice 0)

   mov      vettAuto[0].CodModello, 2243
   mov      vettAuto[0].Cilindrata, 2000
   mov      vettAuto[0].Cavalli, 120
   mov      vettAuto[0].Revisioni[0], 1100000111111011000b
   mov      vettAuto[0].Revisioni[4], 1100000111111011010b
   mov      vettAuto[0].Revisioni[8], 1100000111111011100b

; inizializzazione di vettAuto[46] (elemento di indice 1)

   mov      vettAuto[46].CodModello, 2244
   mov      vettAuto[46].Cilindrata, 2500
   mov      vettAuto[46].Cavalli, 170
   mov      vettAuto[46].Revisioni[0], 1100000111111011000b
   mov      vettAuto[46].Revisioni[4], 1100000111111011010b
   mov      vettAuto[46].Revisioni[8], 1100000111111011100b

; somma a 32 bit tra varDword e DataViaggio

   mov      eax, varDword
   add      eax, DataViaggio

; somma a 16 bit tra varWord e i primi 16 bit di DataViaggio

   mov      ax, varWord
   add      ax, word ptr DataViaggio[0]

; somma a 16 bit tra varWord e i secondi 16 bit di DataViaggio

   mov      ax, varWord
   add      ax, word ptr DataViaggio[2]

; somma a 8 bit tra varByte e i primi 8 bit di DataViaggio

   mov      al, varByte
   add      al, byte ptr DataViaggio[0]

; somma a 8 bit tra varByte e i secondi 8 bit di DataViaggio

   mov      al, varByte
   add      al, byte ptr DataViaggio[1]

; somma a 8 bit tra varByte e i terzi 8 bit di DataViaggio

   mov      al, varByte
   add      al, byte ptr DataViaggio[2]

; somma a 8 bit tra varByte e i quarti 8 bit di DataViaggio
```

```
   mov      al, varByte
   add      al, byte ptr DataViaggio[3]

; somma a 32 bit tra FiatPanda.Revisioni[4] e vettAuto[46].Revisioni[8]

   mov      eax, FiatPanda.Revisioni[4]
   add      eax, vettAuto[46].Revisioni[8]

; somma a 32 bit tra varDword e vettAuto[46].Revisioni[8]

   mov      eax, varDword
   add      eax, vettAuto[46].Revisioni[8]

; somma a 16 bit tra varWord e i primi 16 bit di vettAuto[46].Revisioni[8]

   mov      ax, varWord
   add      ax, word ptr vettAuto[46].Revisioni[8+0]


; somma a 16 bit tra varWord e i secondi 16 bit di vettAuto[46].Revisioni[8]

   mov      ax, varWord
   add      ax, word ptr vettAuto[46].Revisioni[8+2]

; somma a 8 bit tra varByte e i primi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+0]

; somma a 8 bit tra varByte e i secondi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+1]

; somma a 8 bit tra varByte e i terzi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+2]

; somma a 8 bit tra varByte e i quarti 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+3]

; somma a 8 bit tra i secondi 8 bit di FiatPanda.Revisioni[4] e
; i terzi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, byte ptr FiatPanda.Revisioni[4+1]
   add      al, byte ptr vettAuto[46].Revisioni[8+2]

;-------- fine blocco principale istruzioni -------

   mov      ah, 4ch                ; servizio Terminate Program
   mov      al, 00h                ; exit code = 0
   int      21h                    ; chiama i servizi DOS

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)
```

```
STACKSEGM   ENDS

;##################################################

   END      start
```

Il contenuto del programma di Figura 1 e' estremamente semplice e non necessita di ulteriori spiegazioni; inoltre, attraverso i vari commenti, si possono ricavare tutte le informazioni necessarie per capire lo scopo delle varie dichiarazioni, definizioni, linee di codice, etc.
Osserviamo che, per il segmento di codice **CODESEGM**, e' stato scelto un allineamento di tipo **DWORD**; piu' avanti vedremo che questa scelta ci permettera' di analizzare alcuni aspetti molto importanti.

13.2 Installazione dell'assembler

L'installazione dell'assembler sul computer, non presenta nessuna difficolta'; nel caso piu' generale, questa fase si svolge dalla linea di comando (**prompt del DOS**). A tale proposito, molti assembler forniscono un apposito file eseguibile che svolge questo lavoro in modo automatico; durante la fase di installazione, vengono rivolte all'utente semplici domande che riguardano, in particolare, la cartella nella quale disporre i vari strumenti (assembler, linker, debugger, etc).
Nel caso del **TASM**, il file che si occupa dell'installazione si chiama in genere **INSTALL.EXE**; se si ha a disposizione una versione del **TASM** fornita su floppy disk, bisogna innanzi tutto posizionarsi sul primo dischetto, impartendo dal **prompt del DOS** il comando:

```
A:
```

A questo punto, si puo' lanciare l'installazione con il comando:

```
INSTALL
```

Nel caso del **MASM32**, tutta la fase di installazione si svolge dall'interfaccia grafica di **Windows**; a tale proposito, si possono consultare le istruzioni presenti nella sezione **Downloads** di questo sito.

Per comodita', in tutti gli esempi presentati nella sezione **Assembly Base**, si suppone che il **TASM** sia stato installato nella cartella:

```
C:\TASM
```

e che il **MASM** sia stato installato nella cartella:

```
C:\MASM
```

In questo caso, tutti gli strumenti di sviluppo (assembler, linker, debugger, etc), vengono sistemati nelle cartelle:

```
C:\TASM\BIN
```

e:

```
C:\MASM\BIN
```

13.3 Scrittura del codice sorgente

Come gia' sappiamo, per la scrittura del codice sorgente dobbiamo munirci di un apposito **editor** capace di gestire file rigorosamente in formato testo semplice ASCII; qualsiasi altro formato di file, non viene accettato, ne' da **TASM**, ne' da **MASM**.
La scelta dell'editor e' una questione di gusti personali; in generale, e' consigliabile l'uso di un editor rivolto esplicitamente allo sviluppo dei programmi. Infatti, questi particolari editor sono dotati di caratteristiche che si rivelano di grande aiuto per il programmatore; tra le caratteristiche piu' utili si possono citare, il supporto del **copia/incolla**, la gestione delle tabulazioni (per l'indentazione del codice), la possibilita' di interagire facilmente con il **prompt del DOS**, etc.
Si possono utilizzare ad esempio, gli editor forniti in dotazione ai compilatori e agli interpreti **C**,

Pascal, **BASIC**, etc; bisogna pero' prestare particolare attenzione al fatto che alcuni di questi editor (come quelli del **Visual Basic** o del **Quick Basic**), salvano il codice sorgente in un formato binario che risulta illeggibile con altri editor di testo.
Il **MASM32** e' dotato di un ottimo editor integrato e altamente configurabile, che si chiama **QEDITOR.EXE**; per avviarlo, basta un doppio click sull'icona del file **QEDITOR.EXE** che si trova nella cartella:

```
C:\MASM32
```

Un'altra possibilita' consiste nello scaricare da **Internet** uno dei numerosi editor per programmatori; molti di questi editor sono scaricabili e utilizzabili con licenza **freeware**.
In caso di necessita', si rivela validissimo anche l'editor fornito con il **DOS**; a tale proposito, dal **prompt del DOS** bisogna impartire il comando:

```
edit
```

Una volta che abbiamo l'editor a disposizione, possiamo procedere con la scrittura del codice sorgente; a tale proposito, e' sufficiente eseguire un **copia/incolla** dalla Figura 1 all'editor stesso (questo sistema non funziona con l'editor del **DOS**).
Il codice sorgente cosi' ottenuto, deve essere salvato con il nome **EXETEST.ASM**; per convenzione, un file con estensione **ASM** rappresenta un modulo contenente codice sorgente **Assembly**. Ogni linguaggio di programmazione, utilizza alcune estensioni predefinite per i file che contengono il codice sorgente; nel linguaggio **C** si utilizza l'estensione **C**, nel **C++** si utilizza l'estensione **CPP**, nel **Pascal** si utilizza l'estensione **PAS** e cosi' via. L'estensione predefinita per i file che contengono codice sorgente **Assembly** e' appunto **ASM**; queste regole non sono obbligatorie, ma e' sempre meglio rispettarle per poter individuare facilmente i vari tipi di file.

Per comodita', e' opportuno crearsi una cosiddetta **cartella di lavoro**, nella quale disporre il codice sorgente dei propri programmi **Assembly**; in tutti gli esempi presentati nella sezione **Assembly Base**, si suppone che le cartelle di lavoro siano:

```
C:\TASM\ASMBASE
```

e:

```
C:\MASM\ASMBASE
```

13.4 La fase di assemblaggio (assembling)

La fase di assemblaggio viene svolta, ovviamente, da uno strumento chiamato **assembler** (assemblatore); nel caso del **TASM**, l'assembler si chiama **TASM.EXE**, mentre nel caso del **MASM**, l'assembler si chiama **ML.EXE**.
Entrambi gli assembler funzionano dal **prompt del DOS** o **command line** (linea di comando); come sappiamo, se si sta lavorando sotto **Windows** e' necessario aprire una **DOS Box** attraverso il menu:

```
Start - Programmi - Prompt di MS-DOS
```

Per poter interagire al meglio con **Windows**, si consiglia di configurare la **DOS Box** in modalita' **finestra**; per fare questo bisogna:

1) cliccare con il tasto destro del mouse sul menu **Prompt di MS-DOS**;
2) nel menu che si apre, selezionare **Proprieta'**;
3) nella finestra **Proprieta'**, selezionare la sezione **Schermo**;
4) scegliere le opzioni **Finestra**, **Dimensioni Iniziali: 50 righe** e **Visualizza la barra degli strumenti**;
5) premere i pulsanti **Applica** e **OK**.

A questo punto, possiamo lanciare il **Prompt del DOS** in modalita' finestra; per una migliore resa estetica, si consiglia di selezionare il font **8x12** attraverso il pulsante dei font della **DOS Box**

(questo font e' adatto ad una risoluzione dello schermo pari a **1024x768**). Per chiudere la **DOS Box** e' necessario impartire il comando:

```
exit
```

Il compito fondamentale svolto dall'assembler consiste nel convertire il codice sorgente di un modulo **Assembly**, in un formato binario che prende il nome di **object code** (codice oggetto); come vedremo tra breve, il codice oggetto contiene, oltre al codice macchina, anche una numerosa serie di informazioni necessarie al **linker** per poter svolgere la fase successiva.
Per prendere confidenza con gli assembler di **TASM** e **MASM**, dal **prompt del DOS** possiamo impartire i comandi:

```
c:\tasm\bin\tasm /?
```

o:

```
c:\masm\bin\ml /?
```

In questo modo, si ottiene la lista delle opzioni che si possono passare ai rispettivi assembler; attraverso l'uso di queste opzioni, e' possibile pilotare il comportamento dell'assembler in base alle nostre specifiche esigenze.

Procediamo ora con la fase di assemblaggio del modulo **EXETEST.ASM** che, in precedenza, abbiamo salvato nella cartella **ASMBASE**; prima di tutto dobbiamo posizionarci nella stessa cartella **ASMBASE** impartendo dal **prompt del DOS** il comando:

```
cd c:\tasm\asmbase
```

o:

```
cd c:\masm\asmbase
```

Una volta che ci siamo posizionati nella cartella di lavoro, possiamo finalmente impartire i comandi per l'assemblaggio del nostro programma; nel caso del **TASM**, il comando da impartire e':

```
..\bin\tasm /l exetest.asm
```

mentre nel caso del **MASM**, il comando da impartire e':

```
..\bin\ml /c /Fl exetest.asm
```

Premendo ora il tasto **[Invio]**, parte la fase di assemblaggio del modulo **EXETEST.ASM**; in questa fase, l'assembler verifica anche la presenza di eventuali errori nel codice sorgente. Per ogni eventuale errore che viene individuato, l'assembler genera una serie di messaggi che indicano, il tipo di errore, e in quale linea del codice sorgente si e' verificato il problema; in questo caso, bisogna tornare nell'editor per apportare le necessarie correzioni al codice sorgente.
L'assembler individua principalmente due tipi di errore, e cioe', errori **sintattici** ed errori **semantici**; gli errori sintattici sono in pratica gli errori di battitura che vengono commessi dal programmatore. Ad esempio, per sbaglio si puo' scrivere **MOZ** invece di **MOV**; in questo caso, l'assembler genera un messaggio di errore del tipo:

```
Syntax error
```

Gli errori semantici invece, si riferiscono a parti del codice sorgente prive di significato logico; ad esempio, una istruzione di trasferimento dati da **BL** ad **AX** (da **Reg8** a **Reg16**), puo' essere valida dal punto di vista sintattico, ma semanticamente e' priva di senso.
Sempre a proposito di errori, e' necessario sottolineare che nel caso del **TASM**, se si vogliono utilizzare ad esempio le nuove direttive mostrate nella Figura 13 del precedente capitolo, bisogna disporre della versione **5.0** o superiore di questo assembler; in questo modo si ha a disposizione, l'assembler classico **TASM.EXE**, e l'assembler avanzato **TASM32.EXE** che supporta tutte le nuove caratteristiche delle versioni piu' recenti del **MASM**.
Se il nostro codice sorgente non presenta errori, la fase di assemblaggio si conclude con la generazione del file contenente il codice oggetto; nel caso del nostro esempio, in assenza di diverse indicazioni da parte del programmatore, questo file verra' chiamato **EXETEST.OBJ**, dove l'estensione **OBJ** significa appunto **object file**.
Se si prova a caricare questo file binario in un editor di testo, verra' visualizzata una sequenza incomprensibile di simboli appartenenti al set di codici ASCII; gli editor di testo infatti, cercheranno di convertire in stringhe di testo quelli che, in realta', sono i codici macchina delle

istruzioni del nostro programma.

13.4.1 Il Listing File

Impartendo ora dal prompt il comando:
```
dir
```
verra' mostrata la lista dei file presenti nella cartella **ASMBASE**; in questo modo si scopre che, oltre a **EXETEST.ASM** e **EXETEST.OBJ**, e' presente anche un terzo file chiamato **EXETEST.LST**. L'assembler ha generato questo file, perche' noi glielo abbiamo chiesto attraverso una apposita opzione di assemblaggio; si tratta dell'opzione **/l** per il **TASM** e **/Fl** per il **MASM** (l'opzione **/c = compile only**, indica al **MASM** di limitarsi al solo assemblaggio del file **EXETEST.ASM**).
L'opzione **/l** o **/Fl**, significa **generate listing**, e indica all'assembler di generare un particolare file in formato testo ASCII, chiamato **listing file**; questo file ci offre l'eccezionale opportunita' di analizzare in dettaglio tutto il lavoro svolto da **MASM** e **TASM** nella fase di assemblaggio di un modulo **Assembly**. La Figura 2 mostra proprio il contenuto del file **EXETEST.LST** generato da **TASM**, che possiamo visualizzare con un normale editor di testo; l'analogo file generato da **MASM** e' del tutto simile.

Figura 2 - File EXETEST.LST

```
 1
 2                              ;----------------------------------------------;
 3                              ; File  exetest.asm                            ;
 4                              ; Esempio di programmain formato EXE           ;
 5                              ;----------------------------------------------;
 6
 7                              ;########### direttive per l'assembler ############
 8
 9                               .386                             ; set di istruzioni a 32 bit
10
11                               ;######### dichiarazione tipi e costanti ##########
12
13          =0400                STACK_SIZE = 0400h               ; 1024 byte per lo stack
14
15                               ; dichiarazione record      RecData
16
17                               RecData RECORD       Giorno: 5, Mese: 4, Anno: 11
18
19                               ; dichiarazione struttura Rect
20
21 00000000                      Point2d STRUC
22
23 00000000  01*(????)              x     dw    ?
24 00000002  01*(????)              y     dw    ?
25
26 00000004                      Point2d ENDS
27
28 00000000                      Rect STRUC
29
30 00000000  01*(04*(??))           p1    Point2d  < ?,       ? >
31 00000004  01*(04*(??))           p2    Point2d  < ?,       ? >
32
33 00000008                      Rect ENDS
34
35                               ; dichiarazione struttura Automobile
36
37 00000000                      Automobile  STRUC
38
39 00000000  01*(????)              CodModello dw       ?
40 00000002  01*(????)              Cilindrata dw       ?
41 00000004  01*(????)              Cavalli    dw       ?
42 00000006  01*(0A*(00000000))     Revisioni  RecData  10 DUP ( < > )
43
44 0000002E                      Automobile  ENDS
45
```

```
    46                          ;############### segmento dati ##################
    47
    48   0000                   DATASEGM    SEGMENT  PARA PUBLIC USE16     'DATA'
    49
    50                          ;----- inizio definizione variabili  statiche -----
    51
    52   0000  0B               varByte     db      0bh                     ; intero a 1 byte
    53   0001  ABCD             varWord     dw      0abcdh                  ; intero a 2 byte
    54   0003  12345678         varDword    dd      12345678h               ; intero a 4 byte
    55   0007  123456789ABCDEF0    varQword    dq      123456789abcdef0h       ; intero a 6 byte
    56   000F  00001111222233334444 varTbyte    dt      00001111222233334444h   ; intero a 10 byte
    57   0019  4048F5C3         varFloat32  dd              3.14            ; reale a 4 byte
```



```
    58   001D  968F5FBD20E7F0A8    varFloat64  dq       -5.12345E-200        ; reale a 8 byte
    59
    60   0025  000637D1         DataViaggio RecData   < 12, 06, 2001 >    ; RecData a 4 byte
    61   0029  08*(??)          varRect     Rect      < >               ; Rect   a 8 byte
    62   0031  0A*(08*(??))     vettRect    Rect      10 dup ( < > )      ; vettore di 10 Rect
    63   0081  06*(??) 0A*      + FiatPanda   Automobile < >          ; Automobile a 46 byte
    64         (00000000)
    65   00AF  14*(06*(??) 0A*    + vettAuto   Automobile20 dup ( < > ) ; vettore di 20 Automobile
    66         (00000000))
    67
    68                          ;------- fine definizione variabili statiche ------
    69
    70   0447                   DATASEGM    ENDS
    71
    72                          ;############### segmento codice #################
    73
    74   0000                   CODESEGM    SEGMENT  DWORD PUBLIC USE16 'CODE'
    75
    76                             assume   cs: CODESEGM         ; associa CODESEGM      a CS
    77
    78   0000                   start:                           ; entry point
    79
    80   0000  B8 0000s            mov     ax, DATASEGM          ; trasferisce DATASEGM
    81   0003  8E D8                     mov     ds, ax         ; in DS attraverso      AX
    82                             assume   ds: DATASEGM         ; associa DATASEGM      a DS
    83
    84                          ;------- inizio blocco      principale istruzioni ------
    85
    86                          ; inizializzazione di varRect
    87
    88   0005  C7 06       0029r 0DAC        mov     varRect.p1.x, 3500
    89   000B  C7 06       002Br 1F40        mov     varRect.p1.y, 8000
    90   0011  C7 06       002Dr 1AC2        mov     varRect.p2.x, 6850
    91   0017  C7 06       002Fr 25E4        mov     varRect.p2.y, 9700
    92
    93                          ; inizializzazione di vettRect[0] (elemento di    indice 0)
    94
    95   001D  C7 06       0031r 3CF2        mov     vettRect[0].p1.x, 3cf2h
    96   0023  C7 06       0033r 8DAB        mov     vettRect[0].p1.y, 8dabh
    97   0029  C7 06       0035r 5C2B        mov     vettRect[0].p2.x, 5c2bh
    98   002F  C7 06       0037r 66F1        mov     vettRect[0].p2.y, 66f1h
    99
   100                          ; inizializzazione di vettRect[8] (elemento di    indice 1)
   101
   102   0035  C7 06       0039r 93F1        mov     vettRect[8].p1.x, 93f1h
   103   003B  C7 06       003Br 359C        mov     vettRect[8].p1.y, 359ch
   104   0041  C7 06       003Dr 86FF        mov     vettRect[8].p2.x, 86ffh
   105   0047  C7 06       003Fr 12BC        mov     vettRect[8].p2.y, 12bch
   106
   107                          ; inizializzazione di FiatPanda
   108
   109   004D  C7 06       0081r 0DBE        mov     FiatPanda.CodModello, 3518
   110   0053  C7 06       0083r 03E8        mov     FiatPanda.Cilindrata, 1000
   111   0059  C7 06       0085r 002D        mov     FiatPanda.Cavalli,      45
   112
   113                          ; inizializzazione di FiatPanda.Revisioni[0] con la data
   114                          ; 12/3/2008 = 1100b/00011b/11111011000b
```



```
   115
```

```
    116     005F  66| C7 06 0087r     +          mov   FiatPanda.Revisioni[0], 11000001111111011000b
    117          000C1FD8
    118
    119                               ; inizializzazione di FiatPanda.Revisioni[4] con la data
    120                               ; 12/3/2010 = 1100b/00011b/11111011010b
    121
    122     0068  66| C7 06 008Br     +          mov   FiatPanda.Revisioni[4], 11000001111111011010b
    123          000C1FDA
    124
    125                               ; inizializzazione di vettAuto[0] (elemento di    indice 0)
    126
    127     0071  C7 06       00AFr 08C3         mov     vettAuto[0].CodModello, 2243
    128     0077  C7 06       00B1r 07D0         mov     vettAuto[0].Cilindrata, 2000
    129     007D  C7 06       00B3r 0078         mov     vettAuto[0].Cavalli, 120
    130     0083  66| C7 06 00B5r     + mov       vettAuto[0].Revisioni[0], 11000001111111011000b
    131          000C1FD8
    132     008C  66| C7 06 00B9r     + mov       vettAuto[0].Revisioni[4], 11000001111111011010b
    133          000C1FDA
    134     0095  66| C7 06 00BDr     + mov       vettAuto[0].Revisioni[8], 11000001111111011100b
    135          000C1FDC
    136
    137                               ; inizializzazione di vettAuto[46] (elemento di indice    1)
    138
    139     009E  C7 06       00DDr 08C4         mov     vettAuto[46].CodModello, 2244
    140     00A4  C7 06       00DFr 09C4         mov     vettAuto[46].Cilindrata, 2500
    141     00AA  C7 06       00E1r 00AA         mov     vettAuto[46].Cavalli, 170
    142     00B0  66| C7 06 00E3r     +  mov  vettAuto[46].Revisioni[0],  11000001111111011000b
    143          000C1FD8
    144     00B9  66| C7 06 00E7r     +  mov  vettAuto[46].Revisioni[4],  11000001111111011010b
    145          000C1FDA
    146     00C2  66| C7 06 00EBr     +  mov  vettAuto[46].Revisioni[8],  11000001111111011100b
    147          000C1FDC
    148
    149                               ; somma a 32 bit tra varDword e DataViaggio
    150
    151     00CB  66| A1 0003r           mov     eax, varDword
    152     00CF  66| 03 06 0025r        add     eax, DataViaggio
    153
    154                       ; somma a 16 bit tra varWord e i primi 16 bit di DataViaggio
    155
    156     00D4  A1 0001r               mov     ax, varWord
    157     00D7  03 06       0025r              add     ax, word ptr DataViaggio[0]
    158
    159                       ; somma a 16 bit tra varWord e i secondi 16 bit di DataViaggio
    160
    161     00DB  A1 0001r               mov     ax, varWord
    162     00DE  03 06       0027r              add     ax, word ptr DataViaggio[2]
    163
    164                               ; somma a 8 bit tra varByte e i primi 8 bit di    DataViaggio
    165
    166     00E2  A0 0000r               mov     al, varByte
    167     00E5  02 06       0025r              add     al, byte ptr DataViaggio[0]
    168
    169                               ; somma a 8 bit tra varByte e i secondi 8 bit di DataViaggio
    170
    171     00E9  A0 0000r               mov     al, varByte

Turbo Assembler        Version 4.1      25/11/03 22:11:31     Page 4
exetest.asm

    172     00EC  02 06       0026r              add     al, byte ptr DataViaggio[1]
    173
    174                       ; somma a 8 bit tra varByte e i terzi 8 bit di DataViaggio
    175
    176     00F0  A0 0000r               mov     al, varByte
    177     00F3  02 06       0027r              add     al, byte ptr DataViaggio[2]
    178
    179                       ; somma a 8 bit tra varByte e i quarti 8 bit di DataViaggio
    180
    181     00F7  A0 0000r               mov     al, varByte
    182     00FA  02 06       0028r              add     al, byte ptr DataViaggio[3]
    183
    184                ; somma a 32 bit tra FiatPanda.Revisioni[4] e vettAuto[46].Revisioni[8]
    185
    186     00FE  66| A1 008Br           mov     eax, FiatPanda.Revisioni[4]
    187     0102  66| 03 06 00EBr        add     eax, vettAuto[46].Revisioni[8]
    188
    189                               ; somma a 32 bit tra varDword e vettAuto[46].Revisioni[8]
```

```
    190
    191    0107  66| A1 0003r            mov     eax, varDword
    192    010B  66| 03 06 00EBr         add     eax, vettAuto[46].Revisioni[8]
    193
    194              ; somma a 16 bit tra varWord e i primi 16 bit di vettAuto[46].Revisioni[8]
    195
    196    0110  A1 0001r                mov     ax, varWord
    197    0113  03 06       00EBr              add     ax, word ptr vettAuto[46].Revisioni[8+0]
    198
    199
    200              ; somma a 16 bit tra varWord e i secondi 16 bit di vettAuto[46].Revisioni[8]
    201
    202    0117  A1 0001r                mov     ax, varWord
    203    011A  03 06       00EDr              add     ax, word ptr vettAuto[46].Revisioni[8+2]
    204
    205              ; somma a 8 bit tra varByte e i primi 8 bit di vettAuto[46].Revisioni[8]
    206
    207    011E  A0 0000r                mov     al, varByte
    208    0121  02 06       00EBr              add     al, byte ptr vettAuto[46].Revisioni[8+0]
    209
    210              ; somma a 8 bit tra varByte e i secondi 8 bit di vettAuto[46].Revisioni[8]
    211
    212    0125  A0 0000r                mov     al, varByte
    213    0128  02 06       00ECr              add     al, byte ptr vettAuto[46].Revisioni[8+1]
    214
    215              ; somma a 8 bit tra varByte e i terzi 8 bit di vettAuto[46].Revisioni[8]
    216
    217    012C  A0 0000r                mov     al, varByte
    218    012F  02 06       00EDr              add     al, byte ptr vettAuto[46].Revisioni[8+2]
    219
    220              ; somma a 8 bit tra varByte e i quarti 8 bit di vettAuto[46].Revisioni[8]
    221
    222    0133  A0 0000r                mov     al, varByte
    223    0136  02 06       00EEr              add     al, byte ptr vettAuto[46].Revisioni[8+3]
    224
    225              ; somma a 8 bit tra i secondi 8 bit di FiatPanda.Revisioni[4] e
    226              ; i terzi 8 bit di vettAuto[46].Revisioni[8]
    227
    228    013A  A0 008Cr                mov     al, byte ptr FiatPanda.Revisioni[4+1]

Turbo Assembler       Version 4.1      25/11/03 22:11:31     Page 5
exetest.asm

    229    013D  02 06       00EDr              add     al, byte ptr vettAuto[46].Revisioni[8+2]
    230
    231                            ;-------- fine       blocco principale istruzioni -------
    232
    233    0141  B4 4C                          mov     ah, 4ch              ; servizio
       Terminate Program
    234    0143  B0 00                          mov     al, 00h              ; exit code = 0
    235    0145  CD 21                          int     21h                  ; chiama i
       servizi DOS
    236
    237    0147                    CODESEGM    ENDS
    238
    239                            ;################# segmento stack #################
    240
    241    0000                    STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'
    242
    243    0000  0400*(??)                      db       STACK_SIZE dup (?)
    244
    245    0400                    STACKSEGM   ENDS
    246
    247                            ;##################################################
    248
    249                               END     start

Turbo Assembler       Version 4.1      25/11/03 22:11:31     Page 6
Symbol Table

Symbol Name                    Type  Value

??date                         Text  "25/11/03"
??filename                     Text  "exetest "
??time                         Text  "22:11:31"
??version                      Number 040A
@Cpu                           Text  0F0FH
@FileName                      Text  exetest
```

```
@WordSize                      Text   2
@curseg                        Text   STACKSEGM
DataViaggio                    Dword  DATASEGM:0025
FiatPanda                      Struct DATASEGM:0081 Automobile
STACK_SIZE                     Number 0400
start                          Near   CODESEGM:0000
varByte                        Byte   DATASEGM:0000
varDword                       Dword  DATASEGM:0003
varFloat32                     Dword  DATASEGM:0019
varFloat64                     Qword  DATASEGM:001D
varQword                       Qword  DATASEGM:0007
varRect                        Struct DATASEGM:0029 Rect
varTbyte                       Tbyte  DATASEGM:000F
varWord                        Word   DATASEGM:0001
vettAuto                       Struct DATASEGM:00AF Automobile
vettRect                       Struct DATASEGM:0031 Rect

Structure Name                 Type  Offset

Automobile
 CodModello                    Word   0000
 Cilindrata                    Word   0002
 Cavalli                       Word   0004
 Revisioni                     Dword  0006
Point2d
 x                             Word   0000
 y                             Word   0002
Rect
 p1                            Struct 0000 Point2d
 p2                            Struct 0004 Point2d

Record Name                    Width Shift

RecData
 Giorno                        05    0F
 Mese                          04    0B
 Anno                          0B    00

Groups & Segments              Bit Size Align  Combine Class

CODESEGM                       16  0147 Dword  Public  CODE
DATASEGM                       16  0447 Para         Public  DATA
STACKSEGM                      16  0400 Para         Stack   STACK
```

Analizziamo in dettaglio l'importantissimo contenuto di questo file, che ci permette di verificare in pratica, le considerazioni teoriche svolte nei precedenti capitoli.
Notiamo subito che il listing file contiene (nella parte destra) anche il codice sorgente del nostro programma; si tratta di una grande comodita' che ci permette di confrontare facilmente ogni linea del nostro programma, con il corrispondente codice ricavato dall'assembler.
Nella parte sinistra del listing file, troviamo una serie di informazioni molto utili; queste informazioni sono disposte su varie colonne. La colonna piu' a sinistra contiene la numerazione progressiva delle varie linee del codice sorgente; questa numerazione, viene generata a beneficio del programmatore che cosi', puo' avere dei punti di riferimento all'interno del programma che deve analizzare.
La seconda colonna contiene gli offset delle varie informazioni presenti nei segmenti di programma e nelle eventuali dichiarazioni; piu' avanti analizzeremo il procedimento seguito dall'assembler per calcolare questi offset.
La terza colonna del listing file e' sicuramente la piu' importante, in quanto contiene il codice macchina generato dall'assembler; analizziamo piu' da vicino le informazioni contenute in queste colonne.

Come si vede in Figura 2, le varie direttive e dichiarazioni, non hanno nessun codice macchina, e quindi non provocano nessuna allocazione di memoria; come gia' sappiamo infatti, si tratta di semplici disposizioni che il programmatore impartisce all'assembler. La riga **9** ad esempio, contiene la direttiva **.386** che ha il solo scopo di informare l'assembler, sul fatto che vogliamo utilizzare il set di istruzioni a **32** bit della **CPU 80386**; questa direttiva (come qualsiasi altra direttiva) non ha quindi

nessun offset e nessun codice macchina.

La riga **13** contiene la dichiarazione della costante simbolica **STACK_SIZE**; come possiamo notare, l'assembler si limita a rilevare che questo nome simbolico e' associato al valore costante **0400h**. Ogni volta che l'assembler incontra questo nome simbolico in una istruzione o in una espressione costante, lo sostituisce con il valore **0400h** che verra' quindi incorporato direttamente nel codice macchina.

Tra la riga **15** e la riga **44**, notiamo la presenza delle varie dichiarazioni di **RECORD** e **STRUC**; anche in questo caso si vede che l'assembler non genera nessun codice macchina.
Per quanto riguarda le **STRUC**, vediamo che l'assembler calcola gli offset dei vari membri; e' importante ricordare che questi offset, sono calcolati rispetto all'inizio della struttura di appartenenza. In relazione ad esempio alla struttura **Automobile**, l'assembler rileva che:

***** il membro **CodModello** e' formato da **1** word non inizializzata (**01*(????)**), e si trova all'offset **0000h** rispetto all'inizio di **Automobile**;
***** il membro **Cilindrata** e' formato da **1** word non inizializzata (**01*(????)**), e si trova all'offset **0002h** rispetto all'inizio di **Automobile**;
***** il membro **Cavalli** e' formato da **1** word non inizializzata (**01*(????)**), e si trova all'offset **0004h** rispetto all'inizio di **Automobile**;
***** il membro **Revisioni** e' formato da **1** gruppo di **10** dword inizializzate a **0** (**01*(0A*(00000000))**), e si trova all'offset **0006h** rispetto all'inizio di **Automobile**.

Complessivamente, le istanze di tipo **Automobile**, richiedono una quantita' di memoria pari a:
`2 + 2 + 2 + (10 * 4) = 6 + 40 = 46 = 2Eh byte`
E' importante notare che tutti i codici numerici generati dall'assembler (compresi gli offset), sono espressi implicitamente in esadecimale; cio' dimostra l'importanza per il programmatore **Assembly**, della conoscenza del sistema di codifica esadecimale.

13.4.2 Assemblaggio di DATASEGM

A questo punto l'assembler arriva alla linea **48** del listing file, dove incontra l'inizio del segmento di dati **DATASEGM**; per ciascun dato presente in questo segmento, l'assembler procede al calcolo delle tre informazioni fondamentali che sono, come sappiamo, l'offset, la dimensione in byte e il contenuto iniziale.
Il primo aspetto importante da osservare, riguarda il fatto che l'assembler non e' ovviamente in grado di determinare in anticipo l'indirizzo fisico di memoria da cui partira' un determinato segmento di programma; tutto cio' che riguarda il corretto allineamento in memoria dei segmenti di programma, e' di competenza del linker. Proprio per questo motivo, l'assembler assegna ad ogni segmento di programma, un indirizzo fisico iniziale simbolico, che e' sempre allineato al paragrafo; questo indirizzo fisico e' quindi del tipo **XXXX0h**, e puo' essere associato all'indirizzo logico normalizzato **XXXXh:0000h**. In fase di assemblaggio, il valore **XXXXh** assunto dalla componente **Seg** di questo indirizzo, e' del tutto irrilevante; il **TASM** ad esempio, utilizza il valore simbolico **0000h**, mentre il **MASM**, utilizza il valore simbolico **----**. Cio' che conta, e' che l'assembler fa' partire sempre da **0000h** gli offset relativi alle informazioni contenute in un qualunque segmento di programma; in questa fase infatti, lo scopo dell'assembler e' quello di determinare lo spiazzamento che assume ogni informazione, all'interno del segmento di appartenenza.
Per calcolare gli indirizzi **Seg:Offset** delle informazioni presenti in un segmento di programma, l'assembler si serve di uno strumento chiamato **location counter** (contatore di locazione); il location counter viene indicato con il simbolo **$**, e in fase di assemblaggio rappresenta istante per istante, l'offset corrente all'interno del segmento di programma che l'assembler sta esaminando. E'

importante quindi ricordare che **$** individua una coppia **Seg:Offset**; man mano che l'assembler avanza nella scansione del segmento di programma, la componente **Offset** del contatore **$** viene incrementata in modo che segni l'offset corrente.
Ogni volta che l'assembler incontra l'inizio di un nuovo segmento di programma, inizializza **$** in modo che la sua componente **Offset** valga **0000h**; a questo punto inizia il calcolo degli offset relativi alle informazioni presenti nel segmento stesso. Nel caso di **DATASEGM**, il location counter viene inizializzato con **DATASEGM:0000h** (cioe' **XXXXh:0000h**); a questo punto, l'assembler, come si vede in Figura 2, rileva quanto segue:

***** La componente **Offset** di **$** viene inizializzata a **0000h** e si ottiene:
`Offset = 0000h`
Quest'offset viene assegnato a **varByte** che occupa **1** byte (**0001h** byte).
***** La componente **Offset** di **$** viene incrementata di **0001h** e si ottiene:
`Offset = 0000h + 0001h = 0001h`
Quest'offset viene assegnato a **varWord** che occupa **2** byte (**0002h** byte).
***** La componente **Offset** di **$** viene incrementata di **0002h** e si ottiene:
`Offset = 0001h + 0002h = 0003h`
Quest'offset viene assegnato a **varDword** che occupa **4** byte (**0004h** byte).
***** La componente **Offset** di **$** viene incrementata di **0004h** e si ottiene:
`Offset = 0003h + 0004h = 0007h`
Quest'offset viene assegnato a **varQword** che occupa **8** byte (**0008h** byte).
***** La componente **Offset** di **$** viene incrementata di **0008h** e si ottiene:
`Offset = 0007h + 0008h = 000Fh`
Quest'offset viene assegnato a **varTbyte** che occupa **10** byte (**000Ah** byte).
***** La componente **Offset** di **$** viene incrementata di **000Ah** e si ottiene:
`Offset = 000Fh + 000Ah = 0019h`
Quest'offset viene assegnato a **varFloat32** che occupa **4** byte (**0004h** byte).
***** La componente **Offset** di **$** viene incrementata di **0004h** e si ottiene:
`Offset = 0019h + 0004h = 001Dh`
Quest'offset viene assegnato a **varFloat64** che occupa **8** byte (**0008h** byte).
***** La componente **Offset** di **$** viene incrementata di **0008h** e si ottiene:
`Offset = 001Dh + 0008h = 0025h`
Quest'offset viene assegnato a **DataViaggio** che occupa **4** byte (**0004h** byte).
***** La componente **Offset** di **$** viene incrementata di **0004h** e si ottiene:
`Offset = 0025h + 0004h = 0029h`
Quest'offset viene assegnato a **varRect** che occupa **8** byte (**0008h** byte).
***** La componente **Offset** di **$** viene incrementata di **0008h** e si ottiene:
`Offset = 0029h + 0008h = 0031h`
Quest'offset viene assegnato a **vettRect** che occupa **10 * 8 = 80** byte (**0050h** byte).
***** La componente **Offset** di **$** viene incrementata di **0050h** e si ottiene:
`Offset = 0031h + 0050h = 0081h`
Quest'offset viene assegnato a **FiatPanda** che occupa **46** byte (**002Eh** byte).
***** La componente **Offset** di **$** viene incrementata di **002Eh** e si ottiene:
`Offset = 0081h + 002Eh = 00AFh`
Quest'offset viene assegnato a **vettAuto** che occupa **20 * 46 = 920** byte (**0398h** byte).

In un segmento di programma con attributo **USE16**, la componente **Offset** di un indirizzo logico e' un valore a **16** bit; in questo caso, la componente **Offset** di **$** puo' assumere tutti i valori compresi tra **0000h** e **FFFFh**. E' chiaro che, se la componente **Offset** di **$** supera **FFFFh**, ricomincia a contare da **0000h**; in tal caso, l'assembler genera un messaggio di errore del tipo:
`Location counter overflow`

Analizzando i vari offset assegnati ai dati definiti in **DATASEGM**, si puo' notare che l'assembler compatta al massimo le varie informazioni presenti nei segmenti di programma, in modo da non

sprecare neanche un byte; in base allora al fatto che **vettAuto** parte dall'offset **00AFh** e occupa **0398h** byte, possiamo dire che la dimensione totale di **DATASEGM** e' pari a:

```
00AFh + 0398h = 0447h byte = 1095 byte
```

Nella terza colonna di Figura 2, tra le righe **52** e **65**, vengono disposti i valori iniziali di ciascun dato definito in **DATASEGM**; tutte queste informazioni ricavate dall'assembler, vengono inserite nel codice oggetto e messe a disposizione del linker.

Esclusivamente in fase di assemblaggio di un programma, il location counter **$** e' accessibile in lettura al programmatore; nei capitoli successivi vedremo degli esempi pratici sull'utilizzo di questo strumento.

13.4.3 Assemblaggio di CODESEGM

Terminato l'assemblaggio di **DATASEGM**, l'assembler cerca altri segmenti **DATASEGM** eventualmente presenti nel modulo **EXETEST.ASM**; se la ricerca fornisce esito positivo, l'assembler provvede ad assemblare e ad unire tra loro, tutti i **DATASEGM** aventi gli stessi identici attributi. Nel nostro caso, non essendo presente nessun altro **DATASEGM**, l'assembler passa al segmento successivo, e cioe' a **CODESEGM**; il location counter viene nuovamente inizializzato con la coppia **CODESEGM:0000h**. A questo punto, parte l'assemblaggio del blocco **CODESEGM**; come al solito, se vogliamo verificare il lavoro svolto dall'assembler, possiamo analizzare il listing file di Figura 2.

Notiamo subito che in corrispondenza della riga **76** di Figura 2, alla direttiva **ASSUME** non viene associato nessun offset e nessun codice macchina; e' necessario ribadire ancora una volta, che le direttive non devono essere confuse con le istruzioni, e quindi non comportano nessuna allocazione di memoria.
Alla riga **78** viene incontrata l'etichetta **start**; in questo punto, la componente **Offset** di **$** vale **0000h**; questo e' proprio l'offset che viene assegnato a **start**. Anche in questo caso, e' necessario ricordare che le etichette, sono dei semplici "marcatori" che hanno lo scopo di individuare un offset all'interno di un segmento di programma; infatti, in Figura 2 notiamo che l'assembler assegna a **start** l'offset **0000h**, senza generare nessun codice macchina. Tutto cio' significa che le etichette non occupano memoria; osserviamo che in questo caso particolare, l'etichetta **start** indica l'**entry point** del nostro programma.
A dimostrazione del fatto che **start** non occupa memoria, possiamo notare che subito dopo, e' presente l'istruzione:

```
MOV AX, DATASEGM
```

alla quale l'assembler assegna l'offset **0000h**, che e' lo stesso di **start**; questa istruzione e' la prima in assoluto che verra' elaborata dalla **CPU** non appena iniziera' la fase di esecuzione del nostro programma.
Si tratta chiaramente di un trasferimento dati da **Imm16** a **Reg16**; il codice macchina di questa istruzione, e' formato dall'**Opcode 1011_w_Reg**, seguito da **Imm16**. Nel nostro caso, **w=1** (operandi a **16** bit) e **Reg=AX=000b**; si ottiene quindi:

```
Opcode = 10111000b = B8h
```

Il valore immediato e' **DATASEGM**, che simbolicamente vale **0000h**; di conseguenza, l'assembler genera il codice macchina:

```
10111000b 0000000000000000b = B8h 0000h
```

In Figura 2 notiamo che **TASM** indica l'operando **Imm16** con **0000s**; la **s** sta appunto per **segment**. Nel caso invece del listing file prodotto da **MASM**, si vede che l'operando **Imm16** viene indicato con **----R**; si tratta in ogni caso di valori simbolici che verranno poi modificati dal linker. Tutte le componenti **Seg** e **Offset** destinate ad essere modificate dal linker e dal **SO**, vengono definite **relocatables** (rilocabili); piu' avanti vedremo come avviene la fase di rilocazione.

L'istruzione appena esaminata, ha un codice macchina da **3** byte; di conseguenza, l'assembler incrementa di **0003h** la componente **Offset** del location counter, e ottiene:
```
Offset = 0000h + 0003h = 0003h
```
Quest'offset viene assegnato all'istruzione successiva, che e':
```
MOV DS, AX
```
Si tratta chiaramente di un trasferimento dati da **Reg16** a **SegReg**; il codice macchina di questa istruzione, e' formato dall'**Opcode 10001110b=8Eh** e dal campo **mod_0_SegReg_r/m**. Nel nostro caso, **mod=11b** (entrambi gli operandi sono di tipo registro), **r/m=AX=000b** e **SegReg=DS=11b**; si ottiene quindi:
```
mod_0_SegReg_r/m = 11011000b = D8h
```
L'assembler genera allora il codice macchina:
```
10001110b 11011000b = 8Eh D8h
```
Questo codice macchina occupa **2** byte, per cui, l'assembler incrementa di **0002h** la componente **Offset** del location counter, e ottiene:
```
Offset = 0003h + 0002h = 0005h
```
Quest'offset viene assegnato all'istruzione successiva.

Nella parte seguente di **CODESEGM**, sono presenti una serie di trasferimenti di dati e addizioni, che coinvolgono, registri, valori immediati e variabili statiche definite in **DATASEGM**; lo scopo di queste istruzioni, e' quello di mostrare che, simboli apparentemente complessi come ad esempio:
```
vettAuto[0].Revisioni[8]
```
non sono altro che banalissimi **Disp16**.
Consideriamo ad esempio l'istruzione (riga **90** di Figura 2):
```
mov varRect.p2.x, 6850
```
Si tratta di un trasferimento dati da **Imm16** a **Mem16** (osserviamo infatti che **x** e' stato definito come dato di tipo **WORD**); il codice macchina di questa istruzione, e' formato dall'**Opcode 1100011w**, seguito dal campo **mod_000_r/m** (il sottocampo **reg** e' superfluo e vale **000b**), da **Disp16** e da **Imm16**. Nel nostro caso, **w=1** (operandi a **16** bit), per cui:
```
Opcode = 11000111b = C7h
```
La destinazione e' di tipo **Disp16**, per cui **mem=00b** e **r/m=110b**; otteniamo quindi:
```
mod_000_r/m = 00000110b = 06h
```
Nel segmento **DATASEGM** di Figura 2 si nota che **varRect** si trova all'offset **0029h**; considerando il fatto che ogni **Point2d** occupa **4** byte, possiamo dire che **varRect.p2.x** si trova all'offset:
```
Disp16 = 0029h + 0004h + 0000h = 002Dh = 0000000000101101b
```
Il valore immediato e':
```
Imm16 = 6850 = 1AC2h = 0001101011000010b
```
In definitiva, l'assembler genera il codice macchina:
```
11000111b 00000110b 0000000000101101b 0001101011000010b = C7h 06h 002Dh 1AC2h
```
Nell'istruzione che abbiamo appena esaminato, l'identificatore **varRect.p2.x** e' privo della componente **Seg**; come gia' sappiamo, cio' e' possibile grazie alla precedente direttiva:
```
ASSUME DS: DATASEGM
```
In questo modo, l'assembler sa' che tutti gli identificatori definiti in **DATASEGM** (come **varRect.p2.x**), hanno un offset che deve essere associato alla componente **Seg** contenuta in **DS**; naturalmente, e' fondamentale che lo stesso **DS** sia gia' stato inizializzato con **DATASEGM**!
Possiamo dire allora che in questo caso, **varRect.p2.x** viene gestita con un indirizzo di tipo **NEAR**, formato dalla sola componente **Offset** (cioe' dal **Disp16=002Dh**); in fase di elaborazione dell'istruzione, la **CPU** provvede ad associare automaticamente il **Disp16** a **DS**.
Se avessimo scritto invece:

```
mov      ax, DATASEGM           ; trasferisce DATASEGM
mov      es, ax                 ; in ES attraverso AX
assume   es: DATASEGM           ; associa DATASEGM a ES
```

allora l'assembler, all'inizio del precedente codice macchina, avrebbe inserito il prefisso **26h** relativo al registro **ES** (segment override); cio' avviene perche', come sappiamo, **ES** non e' il **SegReg** predefinito per i segmenti di dati. In un caso del genere, la variabile **varRect** viene gestita attraverso un indirizzo di tipo **FAR**, formato da una coppia completa **Seg:Offset** (cioe' **ES:002Dh**); ogni indirizzo **FAR** aumenta di **1** byte le dimensioni del codice macchina, e inoltre, essendo formato da **32** bit, comporta un tempo di elaborazione leggermente piu' lungo rispetto al caso degli indirizzi **NEAR**.

Vediamo un altro esempio (linea **223** di Figura 2) relativo all'istruzione:
```
add al, byte ptr vettAuto[46].Revisioni[8+3]
```
Si tratta di una addizione tra **Reg8** (destinazione) e **Mem8** (sorgente); il nome **Revisioni** indica un dato a **32** bit, ma grazie all'operatore **BYTE PTR**, possiamo dire all'assembler che vogliamo accedere ad una locazione di memoria da **8** bit. Il codice macchina di questa istruzione, e' formato dall'**Opcode 000000dw**, seguito dal campo **mod_reg_r/m** e da un **Disp16**; nel nostro caso, **d=1** (destinazione registro) e **w=0** (operandi a **8** bit), per cui:
```
Opcode = 00000010b = 02h
```
Il registro destinazione e' **reg=AL=000b**; la sorgente e' di tipo **Disp16**, per cui **mod=00b** e **r/m=110b**. Otteniamo quindi:
```
mod_reg_r/m = 00000110b = 06h
```
Nel segmento **DATASEGM** di Figura 2 si nota che **vettAuto** si trova all'offset **00AFh**, mentre il membro **Revisioni** si trova all'offset **0006h** (rispetto all'inizio di ogni struttura **Automobile**); tenendo conto del fatto che **46=002Eh**, possiamo dire allora che **vettAuto[46].Revisioni[8+3]** si trova all'offset:
```
Disp16 = 00AFh + 002Eh + 0006h + 0008h + 0003h = 00EEh = 0000000011101110b
```
In definitiva, l'assembler genera il codice macchina:
```
00000010b 00000110b 0000000011101110b = 02h 06h 00EEh
```

Vediamo infine un esempio (linea **151** di Figura 2) relativo all'istruzione:
```
mov eax, varDword
```
Questa volta abbiamo a che fare con un trasferimento dati a **32** bit da **Mem32** a **Reg32**; grazie alla presenza dell'accumulatore, viene utilizzato un codice macchina compatto formato dal solo **Opcode 1010000w** seguito dal **Disp16**. Nel **modo 32 bit**, il bit **w=1** indica che gli operandi sono a **32** bit (full size); otteniamo allora:
```
Opcode = 10100001b = A1h
```
In Figura 2 si nota che **varDword** si trova all'offset **0003h** del segmento **DATASEGM**; otteniamo quindi:
```
Disp16 = 0003h = 0000000000000011b
```
L'assembler genera anche il prefisso **66h** (**01100110b**) che rappresenta l'operand size prefix; alla fine, otteniamo il codice macchina:
```
01100110b 10100001b 0000000000000011b = 66h A1h 0003h
```

Come si puo' notare in Figura 2, complessivamente, l'intero contenuto di **CODESEGM** occupa **0147h** byte di memoria, cioe' **327** byte.

Un'ultima considerazione riguarda il fatto che, come si vede in Figura 2, l'assembler aggiunge una **r** ad ogni **Disp16** presente nelle istruzioni del segmento di codice; anche in questo caso, questa **r** sta per **relocatable**, e indica il fatto che, tutti questi **Disp16**, sono soggetti ad una eventuale rilocazione da parte del linker. Piu' avanti verra' chiarito questo importantissimo aspetto.

13.4.4 Assemblaggio di STACKSEGM

Terminato l'assemblaggio di **CODESEGM**, l'assembler cerca altri segmenti **CODESEGM** eventualmente presenti nel modulo **EXETEST.ASM**; se la ricerca fornisce esito positivo, l'assembler provvede ad assemblare e ad unire tra loro, tutti i **CODESEGM** aventi gli stessi identici attributi. Nel nostro caso, non essendo presente nessun altro **CODESEGM**, l'assembler passa al segmento successivo, e cioe' a **STACKSEGM**; il location counter viene nuovamente inizializzato con la coppia **STACKSEGM:0000h**. A questo punto, parte l'assemblaggio del blocco **STACKSEGM**; trattandosi di un segmento contenente esclusivamente dati, la situazione e' perfettamente analoga al caso di **DATASEGM**.
Come possiamo notare in Figura 2, l'assembler assegna l'offset **0000h** all'unico dato presente, che e' un vettore formato da **0400h** byte non inizializzati (**0400*(??)**); di conseguenza, la dimensione complessiva del segmento **STACKSEGM** e' pari proprio a **0400h** byte, cioe' **1024** byte.

Terminato l'assemblaggio di **STACKSEGM**, l'assembler cerca altri segmenti **STACKSEGM** eventualmente presenti nel modulo **EXETEST.ASM**; se la ricerca fornisce esito positivo, l'assembler provvede ad assemblare e ad unire tra loro, tutti gli **STACKSEGM** aventi gli stessi identici attributi. Nel nostro caso, non essendo presente nessun altro **STACKSEGM**, l'assembler passa al segmento successivo; non essendo presente nessun segmento successivo, la fase di assemblaggio puo' considerarsi terminata.

13.4.5 La Symbol Table

La parte finale del file **EXETEST.LST**, contiene una sezione che prende il nome di **Symbol Table** (tavola dei simboli); la **Symbol Table** ha una importanza enorme in quanto rappresenta un riassunto di tutte le informazioni fondamentali che l'assembler ha ricavato dal codice sorgente del nostro programma.

Come si puo' notare in Figura 2, la parte iniziale della **Symbol Table** contiene una lista chiamata **Symbol Name**; si tratta dell'elenco di tutti gli identificatori presenti nel nostro programma. Alcune di queste informazioni indicano, la data e l'ora di assemblaggio, il nome del modulo sorgente appena assemblato, la versione dell'assembler, etc; queste informazioni variano da assembler ad assembler, e sono riservate all'uso interno dello stesso assembler e del linker.
La parte piu' importante di questa lista, contiene tutti i nomi simbolici dei dati che abbiamo definito nel nostro programma; a ciascun nome, l'assembler associa due informazioni chiamate, **Type** e **Value**. Nel caso ad esempio delle costanti simboliche come **STACK_SIZE**, l'assembler raccoglie le seguenti informazioni:

```
STACK_SIZE Number 0400h
```

Ad ogni costante simbolica, viene quindi associato il tipo **Number** e il valore numerico della costante stessa.
Nel caso dei dati elementari come **varFloat64**, l'assembler raccoglie le seguenti informazioni:

```
varFloat64 Qword DATASEGM:001Dh
```

Ad ogni dato elementare, viene quindi associata la dimensione in byte, e l'indirizzo logico completo **Seg:Offset**.
Nel caso dei dati complessi come **FiatPanda**, l'assembler raccoglie le seguenti informazioni:

```
FiatPanda Struct DATASEGM:0081h Automobile
```

Ad ogni dato strutturato, viene quindi associato il tipo **Struct**, e l'indirizzo logico completo **Seg:Offset** seguito dall'identificatore della dichiarazione di struttura.
Nel caso delle etichette come **start**, l'assembler raccoglie le seguenti informazioni:

```
start Near CODESEGM:0000h
```

Ad ogni etichetta, viene quindi associato il tipo di indirizzo (**NEAR** o **FAR**) individuato, e l'indirizzo logico completo **Seg:Offset**; nel caso di **start**, il tipo **NEAR** indica che per accedere a

questa etichetta dall'interno di **CODESEGM**, basta specificare un indirizzo formato dalla sola componente **Offset** (la componente **Seg** e' implicitamente **CS=CODESEGM**).

Dopo la **Symbol Name**, troviamo una seconda lista chiamata **Structure Name**; come si nota in Figura 2, questa lista contiene l'elenco delle varie dichiarazioni di struttura. Per ciascuna struttura, vengono descritte in dettaglio le caratteristiche interne; queste informazioni comprendono, i nomi dei membri, l'ampiezza in byte di ciascun membro e l'offset interno di ciascun membro (espresso in byte).

Dopo la **Structure Name**, troviamo una terza lista chiamata **Record Name**; questa lista contiene le caratteristiche interne di tutti i **RECORD** che abbiamo dichiarato nel nostro programma. Nel nostro caso, e' presente la descrizione completa di **RecData**; questa descrizione comprende, il nome di ogni membro, l'ampiezza in bit di ogni membro e l'offset interno di ogni membro (espresso in bit).

La **Symbol Table** termina con una lista chiamata **Segments & Groups**; questa lista comprende le informazioni generali relative a tutti i segmenti di programma presenti nel modulo **EXETEST.ASM**. Nel caso di Figura 2, la lista ricavata dall'assembler e' la seguente:

```
Groups & Segments     Bit     Size    Align   Combine Class

CODESEGM              16      0147    Dword   Public  CODE
DATASEGM              16      0447    Para    Public  DATA
STACKSEGM             16      0400    Para    Stack   STACK
```

Il campo **Bit** si riferisce alla dimensione in bit degli offset utilizzati per accedere ai vari segmenti di programma; nel nostro caso, questa dimensione e' **16** in quanto abbiamo utilizzato per tutti i segmenti l'attributo **USE16**.

Un'ultima considerazione riguarda il fatto che, come si nota in Figura 2, tutti i nomi simbolici presenti nella **Symbol Table** (compresi i nomi dei segmenti di programma), risultano disposti in ordine alfabetico; questa disposizione e' solo una convenzione formale usata dell'assembler, e non ha niente a che vedere con la disposizione in memoria dei dati o dei segmenti di programma.

13.5 La fase di linkaggio (linking)

Come molti avranno intuito, il file **EXETEST.OBJ**, pur contenendo il codice macchina del nostro programma, non e' ancora in formato eseguibile; il compito di convertire il codice oggetto in codice eseguibile, spetta ad un altro strumento chiamato **linker** (collegatore), che e' proprio il destinatario delle informazioni generate dall'assembler.

Teoricamente, il contenuto di un file oggetto dovrebbe rispecchiare una struttura standard, indicata con la sigla **OMF** (**Relocatable Object Module Format**); in realta', i vari assembler producono dei file oggetto che, pur seguendo questo standard, si differenziano tra loro per alcuni dettagli. I file oggetto prodotti da **MASM** e **TASM**, danno l'impressione di essere compatibili tra loro; in ogni caso, si consiglia di utilizzare i linker della **Borland** per gli object file prodotti da **TASM**, e i linker della **Microsoft** per gli object file prodotti da **MASM**.
La **Microsoft** ha definito un nuovo standard per gli object file, indicato dalla sigla **COFF** (**Common Object File Format**); il formato **COFF** viene utilizzato da tutti gli strumenti di sviluppo della **Microsoft** (**MASM32**, **Visual C++**, etc), per la realizzazione di applicazioni **Windows**. Il formato **COFF** non viene supportato dagli assembler e compiler della **Borland**; chi volesse studiare in dettaglio gli standard **OMF** e **COFF**, puo' scaricare la documentazione ufficiale dalla pagina dei **Downloads** di questo sito (sezione **Documentazione**).

Un altro aspetto importante, riguarda il fatto che spesso, i linker generano eseguibili destinati ad uno specifico **SO**; nel nostro caso quindi, dobbiamo munirci di un linker capace di generare eseguibili destinati a girare sotto **DOS**. Tutte le versioni del **TASM**, forniscono un linker chiamato **TLINK.EXE**, in grado di generare eseguibili per il **DOS**; nel caso invece del **MASM**, bisogna distinguere tra le varie versioni di questo assembler. Infatti, il linker (**LINK.EXE**) fornito con il **MASM32**, e' in grado di generare eseguibili destinati esplicitamente all'ambiente **Windows** a **32** bit; invece, i linker (**LINK.EXE**) forniti con le versioni di **MASM** precedenti la **6.x**, sono tutti in grado di generare eseguibili per il **DOS**. Se si ha a disposizione solo il **MASM32**, si puo' scaricare il linker per il **DOS** dalla pagina dei **Downloads** di questo sito (sezione **Strumenti di sviluppo**); nella stessa pagina, sono presenti anche le istruzioni di installazione.
Alternativamente, e' anche possibile utilizzare i linker dei vari compilatori **C/C++**, **Pascal**, **FORTRAN**, **BASIC**, etc, destinati all'ambiente **DOS**.

Come e' stato spiegato nei precedenti capitoli, nel caso piu' generale un programma **Assembly** e' formato da due o piu' moduli di codice sorgente; per ciascuno di questi moduli, l'assembler crea il relativo object file. Il compito del linker e' innanzi tutto quello di collegare tra loro i vari moduli; da cio' deriva appunto il nome di collegatore. In questo modo il linker ottiene un unico modulo, all'interno del quale i vari segmenti di programma vengono combinati tra loro secondo le disposizioni impartite dal programmatore; i segmenti risultanti dalle varie combinazioni, vengono disposti secondo i criteri di ordinamento e allineamento in memoria stabiliti ugualmente dal programmatore.
Come si puo' facilmente intuire, in seguito alle varie combinazioni e ai successivi allineamenti, puo' rendersi necessaria la modifica delle componenti **Seg:Offset** relative alle informazioni presenti nei segmenti di programma; questo lavoro di modifica prende il nome di **rilocazione**, e rappresenta sicuramente il compito piu' importante e delicato svolto dal linker.
Una volta che il linker ha portato a termine tutto questo lavoro, viene prodotto un file in formato eseguibile; come vedremo tra breve, nel caso piu' generale, un file eseguibile e' costituito da due parti principali chiamate, **header** (intestazione), e **modulo caricabile** (codice, dati e stack).

Dopo queste importanti precisazioni, possiamo procedere con la generazione del formato eseguibile del nostro programma; a tale proposito, prima di tutto dobbiamo posizionarci nella cartella di lavoro **ASMBASE**. A questo punto, nel caso del **TASM** bisogna impartire il seguente comando:
```
..\bin\tlink exetest.obj
```
Nel caso del **MASM**, il comando da impartire e':
```
..\bin\link /map exetest.obj
```
Premendo il tasto **[Invio]**, parte la fase di linking del file **EXETEST.OBJ**; anche in questa fase, e' possibile che vengano generati dei messaggi di errore (errori del linker), che verranno analizzati nel seguito del capitolo e nei capitoli successivi.
Nel caso del **MASM**, il linker chiede al programmatore di inserire i nomi da assegnare ai vari file che verranno prodotti; premendo **[Invio]** per ogni richiesta, si accettano i nomi predefiniti proposti dal linker.

Nel dare inizio alla fase di linking, il linker non e' ovviamente in grado di conoscere in anticipo l'indirizzo fisico di memoria da cui verra' fatto partire il nostro programma; questa informazione infatti, verra' decisa dal **SO** al momento caricare ed eseguire il programma stesso. Proprio per questo motivo, il linker utilizza simbolicamente l'indirizzo fisico iniziale **00000h**; a questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **0000h:0000h**. Possiamo dire quindi che, al primo segmento del nostro programma, viene assegnato l'indirizzo logico iniziale **0000h:0000h**; osserviamo che questo indirizzo, e' compatibile con qualsiasi attributo di allineamento.
Il linker esamina i vari segmenti di programma, nello stesso ordine stabilito dal programmatore; nel caso del nostro esempio (ordinamento predefinito di tipo sequenziale), il linker esamina, prima

DATASEGM, poi **CODESEGM** e infine **STACKSEGM**.

13.5.1 Linking di DATASEGM

Il primo segmento esaminato dal linker e' **DATASEGM**; a questo segmento viene assegnato l'indirizzo fisico iniziale **00000h** (compatibile con l'attributo **PARA**), a cui corrisponde l'indirizzo logico normalizzato **0000h:0000h**. Di conseguenza, il linker ricava:

```
DATASEGM = 0000h
```

Gli offset all'interno di **DATASEGM**, partono dal valore iniziale **0000h**, proprio come era stato determinato dall'assembler; di conseguenza, tutti i calcoli effettuati dall'assembler vengono confermati (non e' necessaria nessuna rilocazione degli offset dei dati).
Il linker va' alla ricerca di altri **DATASEGM** presenti in altri moduli; nel nostro caso, il programma e' formato da un unico modulo, per cui il linker genera la struttura finale di **DATASEGM**. In base alle informazioni ricavate dall'assembler in Figura 2, questa struttura e' rappresentata da un blocco da **0447h** byte (**1095** byte); i vari byte, occupano tutti gli indirizzi fisici compresi tra **00000h** e:

```
00000h + (0447h - 1h) = 00446h
```

In termini di indirizzi logici, il blocco **DATASEGM** viene inserito nel segmento di memoria n. **0000h**, e occupa tutti gli offset di questo segmento, compresi tra **0000h** e:

```
0000h + (0447h - 1h) = 0446h
```

In sostanza, **DATASEGM** e' compreso tra gli indirizzi logici **0000h:0000h** e **0000h:0446h**; all'interno di questo blocco, il linker inserisce tutti i valori della terza colonna di Figura 2, compresi tra le righe **52** e **65**. La Figura 3, illustra una parte della struttura finale di **DATASEGM**.

Figura 3 - Blocco DATASEGM

```
0B
ABCD
12345678
123456789ABCDEF0
00001111222233334444
4048F5C3
968F5FBD20E7F0A8
000637D1
....................
```

Naturalmente, quando **DATASEGM** verra' caricato in memoria, i vari dati risulteranno disposti in modo consecutivo e contiguo, secondo lo schema di Figura 4:

Figura 4 - DATASEGM in memoria

Contenuto	0Bh	CDh	ABh	78h	56h	34h	12h	F0h	DEh	BCh	9Ah	78h	56h	...
Offset	0000h	0001h	0002h	0003h	0004h	0005h	0006h	0007h	0008h	0009h	000Ah	000Bh	000Ch	.....

In questo schema, gli indirizzi di memoria crescono da sinistra verso destra; di conseguenza, i dati di tipo **WORD**, **DWORD** etc, appaiono disposti al contrario rispetto alla loro rappresentazione con il sistema posizionale. Ad esempio, nel caso del dato **ABCDh**, vediamo che il **BYTE** meno significativo **CDh**, si trova alla sinistra del **BYTE** piu' significativo **ABh**; e' importante abituarsi anche a questo tipo di rappresentazione, in quanto, molti **debugger** e **editor esadecimali**, visualizzano la memoria con gli indirizzi crescenti da sinistra verso destra.
La **CPU** non incontra nessuna difficolta' nel gestire la situazione, apparentemente incomprensibile, di Figura 4; infatti, ogni istruzione che accede a questi dati, codifica l'indirizzo **Seg:Offset** e l'ampiezza in bit dei dati stessi.

13.5.2 Linking di CODESEGM

Terminato il linking di **DATASEGM**, il linker va' alla ricerca di altri segmenti di classe **'DATA'**; nel nostro caso, non esistono altri segmenti di classe **'DATA'**, per cui il linker passa al segmento **CODESEGM**.
Per **CODESEGM** abbiamo richiesto un allineamento di tipo **DWORD**; di conseguenza, il linker calcola un indirizzo fisico iniziale successivo al blocco **DATASEGM**, e multiplo intero di **4**. Abbiamo visto che il blocco **DATASEGM** termina all'indirizzo fisico **00446h**; l'indirizzo fisico successivo a **00446h** e multiplo intero di **4** e' **00448h**. Il linker allora, inserisce un byte di valore **00h** (buco di memoria) all'indirizzo fisico **00447h** in modo da far partire **CODESEGM** da **00448h**; a questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **0044h:0008h**. Di conseguenza, il linker ricava:
```
CODESEGM = 0044h
```
Gli offset all'interno di **CODESEGM**, partono dal valore iniziale **0008h**; questo significa che gli offset calcolati dall'assembler, per tutte le istruzioni di Figura 2, vengono traslati in avanti di **8** byte dal linker. Ad esempio, l'istruzione alla linea **81** di Figura 2:
```
MOV DS, AX
```
passa dall'offset **0003h** all'offset:
```
0003h + 0008h = 000Bh
```

Se al posto di **CODESEGM** ci fosse stato **DATASEGM**, sarebbe successa la stessa cosa; in sostanza, se **DATASEGM** fosse partito dall'indirizzo logico normalizzato **0044h:0008h**, allora:

***** il dato **varByte** sarebbe stato spostato all'offset:
```
0000h + 0008h = 0008h
```
***** il dato **varWord** sarebbe stato spostato all'offset:
```
0001h + 0008h = 0009h
```
***** il dato **varDword** sarebbe stato spostato all'offset:
```
0003h + 0008h = 000Bh
```
***** il dato **varQword** sarebbe stato spostato all'offset:
```
0007h + 0008h = 000Fh
```
e cosi' via.

In una situazione del genere, il linker avrebbe dovuto modificare tutti i **Disp16** presenti nelle istruzioni del programma; nel caso ad esempio dell'istruzione (riga **166** di Figura 2):
```
mov al, varByte
```
il codice macchina:
```
A0h 0000h
```
sarebbe stato trasformato dal linker in:
```
A0h 0008h
```
Le considerazioni appena esposte illustrano il concetto importantissimo di **rilocazione degli offset**.

Tornando al linking di **CODESEGM**, il linker va' alla ricerca di altri **CODESEGM** presenti in altri moduli; nel nostro caso, il programma e' formato da un unico modulo, per cui il linker genera la struttura finale di **CODESEGM**. In base alle informazioni ricavate dall'assembler in Figura 2, questa struttura e' rappresentata da un blocco da **0147h** byte (**327** byte); i vari byte, occupano tutti gli indirizzi fisici compresi tra **00448h** e:
```
00448h + (0147h - 1h) = 0058Eh
```
In termini di indirizzi logici, il blocco **CODESEGM** viene inserito nel segmento di memoria n. **0044h**, e occupa tutti gli offset di questo segmento, compresi tra **0008h** e:
```
0008h + (0147h - 1h) = 014Eh
```
In sostanza, **CODESEGM** e' compreso tra gli indirizzi logici **0044h:0008h** e **0044h:014Eh**; all'interno di questo blocco, il linker inserisce tutti i codici macchina della seconda colonna di

Figura 2, compresi tra le righe **80** e **235**. La Figura 5, illustra una parte della struttura finale di **CODESEGM**.

Figura 5 - Blocco CODESEGM

```
B8 0000
8E D8
C7 06 0029 0DAC
C7 06 002B 1F40
C7 06 002D 1AC2
C7 06 002F 25E4
...............
```

Naturalmente, quando **CODESEGM** verra' caricato in memoria, i vari codici risulteranno disposti in modo consecutivo e contiguo, secondo lo schema di Figura 6:

Figura 6 - CODESEGM in memoria

Contenuto	B8h	00h	00h	8Eh	D8h	C7h	06h	29h	00h	ACh	0Dh	C7h	06	...
Offset	0008h	0009h	000Ah	000Bh	000Ch	000Dh	000Eh	000Fh	0010h	0011h	0012h	0013h	0014h	.....

Come al solito, se la memoria viene disegnata con gli indirizzi crescenti da sinistra verso destra, i dati di tipo **WORD**, **DWORD** etc, appaiono disposti al contrario rispetto alla loro rappresentazione con il sistema posizionale.
Anche in questo caso, la **CPU** non ha nessuna difficolta' a gestire tutti questi codici binari; la **CPU** infatti, dopo aver letto e decodificato il primo byte di una istruzione, e' in grado di sapere se il codice macchina e' terminato, o se bisogna procedere con la lettura e la decodifica di ulteriori byte.

In base al fatto che **CODESEGM** parte dall'indirizzo logico **0044h:0008h**, il linker e' ora in grado di determinare l'indirizzo logico **Seg:Offset** dell'**entry point** del programma; questo indirizzo verra' poi assegnato dal **DOS**, alla coppia **CS:IP**. Nel nostro caso, si tratta ovviamente dell'indirizzo logico assegnato all'etichetta **start**; per questa etichetta, l'assembler aveva calcolato (in Figura 2) una componente **Offset** pari a **0000h**. Il linker ha incrementato di **8** byte quest'offset, per cui, l'indirizzo logico di **start** e' proprio **0044h:0008h**; di conseguenza, il linker ottiene:

```
Entry Point = 0044h:0008h
```

Piu' avanti verra' illustrato il metodo che permette al linker di passare l'**entry point** al **DOS**.

13.5.3 Linking di STACKSEGM

Terminato il linking di **CODESEGM**, il linker va' alla ricerca di altri segmenti di classe **'CODE'**; nel nostro caso, non esistono altri segmenti di classe **'CODE'**, per cui il linker passa al segmento **STACKSEGM**.
Per **STACKSEGM** abbiamo richiesto un allineamento di tipo **PARA**; di conseguenza, il linker calcola un indirizzo fisico iniziale successivo al blocco **CODESEGM**, e multiplo intero di **16**. Abbiamo visto che il blocco **CODESEGM** termina all'indirizzo fisico **0058Eh**; l'indirizzo fisico successivo a **0058Eh** e multiplo intero di **16** e' **00590h**. Il linker allora, inserisce un byte di valore **00h** (buco di memoria) all'indirizzo fisico **0058Fh** in modo da far partire **STACKSEGM** da **00590h**; a questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **0059h:0000h**. Di conseguenza, il linker ricava:

```
STACKSEGM = 0059h
```

Gli offset all'interno di **STACKSEGM**, partono dal valore iniziale **0000h**, proprio come era stato determinato dall'assembler; di conseguenza, tutti i calcoli effettuati dall'assembler vengono confermati (non e' necessaria nessuna rilocazione degli offset dei dati).

Il linker va' alla ricerca di altri **STACKSEGM** presenti in altri moduli; nel nostro caso, il programma e' formato da un unico modulo, per cui il linker genera la struttura finale di **STACKSEGM**. In base alle informazioni ricavate dall'assembler in Figura 2, questa struttura e' rappresentata da un blocco da **0400h** byte (**1024** byte); i vari byte, occupano tutti gli indirizzi fisici compresi tra **00590h** e:
`00590h + (0400h - 1h) = 0098Fh`
In termini di indirizzi logici, il blocco **STACKSEGM** viene inserito nel segmento di memoria n. **0059h**, e occupa tutti gli offset di questo segmento, compresi tra **0000h** e:
`0000h + (0400h - 1h) = 03FFh`
In sostanza, **STACKSEGM** e' compreso tra gli indirizzi logici **0059h:0000h** e **0059h:03FFh**; all'interno di questo blocco, il linker inserisce tutti i valori della terza colonna, riga **243** di Figura 2. Si tratta in pratica di **1024** byte non inizializzati; quest'area da **1024** byte e' destinata ai dati temporanei del programma.

A questo punto, il linker nota la presenza dell'attributo di combinazione **STACK** per il segmento **STACKSEGM**; di conseguenza, il linker procede al calcolo dell'indirizzo logico iniziale **Seg:Offset** che il **DOS** assegnera' poi alla coppia **SS:SP**. Abbiamo visto in precedenza, che il blocco **STACKSEGM** parte dall'indirizzo logico **0059h:0000h** ed occupa **0400h** byte; in base a queste informazioni, il linker ricava facilmente l'indirizzo logico **0059h:0400h** da passare al **DOS**. Se non utilizziamo l'attributo **STACK** per **STACKSEGM**, il compito di inizializzare **SS:SP** (in fase di esecuzione) spetta a noi; in tal caso, dobbiamo scrivere subito dopo l'**entry point**:

```
cli                         ; disabilita le INT. mascherabili
mov   ax, STACKSEGM         ; trasferisce STACKSEGM
mov   ss, ax                ; nel registro SS
mov   sp, 0400h             ; posiziona SP
sti                         ; riabilita le INT. mascherabili
```

La direttiva:
`assume ss: STACKSEGM`
non e' necessaria, a meno che non si voglia accedere per nome a eventuali dati statici definiti nel blocco **STACKSEGM**; come vedremo nei capitoli successivi, e' sconsigliabile definire dati statici nel blocco stack di un programma.

Terminato il linking di **STACKSEGM**, il linker va' alla ricerca di altri segmenti di classe **'STACK'**; nel nostro caso, non esistono altri segmenti di classe **'STACK'**, per cui, la fase di linking dei vari segmenti di programma puo' considerarsi terminata.

13.5.4 Il Map File

Una volta che la fase di linking e' terminata, digitando dal prompt il comando:
`dir`
e premendo **[Invio]**, viene visualizzato, come al solito, l'elenco di tutti i file presenti nella cartella **ASMBASE**; in questo modo, si scopre che, oltre ai **3** file generati dall'assembler, ci sono anche due nuovi file generati dal linker, con i nomi **EXETEST.EXE** e **EXETEST.MAP**.
Naturalmente, **EXETEST.EXE** rappresenta la versione eseguibile del nostro programma; l'estensione **EXE** sta infatti per **executable**. Il file **EXETEST.MAP** invece, rappresenta il cosiddetto **map file**; il suo scopo e' quello di mostrare un riassunto semplificato di tutto il lavoro che il linker ha compiuto sul file oggetto **EXETEST.OBJ**.
Per generare il **map file** con il linker di **MASM**, bisogna utilizzare l'opzione **/map**; nel caso di **TASM** invece, il **map file** viene generato automaticamente (per dire al linker di **TASM** di non generare il **map file**, bisogna passargli l'opzione **/x**).
Il **map file** ci permette di verificare la correttezza di tutti i calcoli che abbiamo effettuato in

precedenza; trattandosi di un file di testo, lo possiamo visualizzare con un normale editor ASCII. La Figura 7, illustra il **map file** generato dal linker del **MASM**; questo file e' del tutto simile a quello generato dal linker del **TASM**.

Figura 7 - File EXETEST.MAP

```
Start  Stop   Length Name        Class

00000H 00446H 00447H DATASEGM    DATA
00448H 0058EH 00147H CODESEGM    CODE
00590H 0098FH 00400H STACKSEGM   STACK

Program entry point at 0044:0008
```

L'esame del file **EXETEST.MAP** conferma l'esattezza di tutti i calcoli relativi agli indirizzi fisici e logici, che abbiamo effettuato in precedenza; osserviamo inoltre che il linker, ha rispettato tutte le direttive che avevamo impartito all'assembler, in relazione all'ordine con il quale disporre in memoria i vari segmenti di programma.
In Figura 7 si nota che anche il linker, come l'assembler, mostra tutte le informazioni numeriche, in formato esadecimale; a maggior ragione quindi, e' importantissimo che i programmatori **Assembly** abbiano grande dimestichezza con la rappresentazione dei numeri in base **16**.

13.6 Struttura di un file EXE

Tra i vari formati eseguibili disponibili per il **DOS**, il formato **EXE** e' in assoluto il piu' potente e flessibile; la caratteristica piu' evidente del formato **EXE**, e' rappresentata dalla possibilita' di inserire al suo interno un numero arbitrario di segmenti di programma. In caso di necessita' quindi, si possono scrivere programmi **EXE** formati da numerosi segmenti di dati e di codice; il numero dei vari segmenti presenti in un programma, puo' crescere sino ad occupare tutta la memoria convenzionale disponibile.

Naturalmente, per poter garantire tutta questa flessibilita', il **DOS** ha bisogno di conoscere una serie di importanti informazioni, relative alla struttura interna del programma **EXE**; in particolare, a causa della presenza di due o piu' segmenti di programma, il **DOS** deve sapere quale di questi segmenti contiene l'**entry point** (per poter inizializzare la coppia **CS:IP**), e quale di questi segmenti verra' usato come stack (per poter inizializzare la coppia **SS:SP**).
Come abbiamo appena visto, gli indirizzi logici iniziali da assegnare alle coppie **CS:IP** e **SS:SP**, sono gia' stati calcolati dal linker; non bisogna dimenticare pero' che il linker, assegna a qualsiasi programma, un indirizzo fisico iniziale simbolico, pari a **00000h**. Come si vede nella Figura 7 del Capitolo 9, i primi Kb della **RAM** sono riservati a svariate informazioni di sistema (vettori di interruzione, area dati del **BIOS**, kernel del **DOS**, etc); e' assolutamente impossibile quindi, che un programma possa essere caricato in memoria a partire dall'indirizzo fisico **00000h**!
Questo significa che anche il **DOS**, dovra' procedere alla rilocazione degli indirizzi logici calcolati dal linker; questa volta, la rilocazione riguarda solo le componenti **Seg** assegnate a questi indirizzi. Infatti, le componenti **Offset** delle informazioni presenti nei segmenti di programma, non sono altro che degli spiazzamenti calcolati rispetto all'inizio dei segmenti stessi; questi spiazzamenti sono gia' stati calcolati (ed eventualmente rilocati) in modo definitivo dal linker, e non devono essere piu' modificati.
Consideriamo, ad esempio, una informazione che si trova all'indirizzo logico **03B2h:0008h**; se ora rilochiamo la componente **Seg** sommandogli il valore **210Ah**, otteniamo il nuovo indirizzo **24BCh:0008h**. Come si puo' notare, la componente **Offset** rimane invariata; in sostanza, l'informazione dell'esempio, e' passata dall'offset **0008h** del segmento di memoria n. **03B2h**, all'offset **0008h** del segmento di memoria n. **24BCh**.

Come conseguenza della rilocazione dei segmenti, il **DOS** ha anche bisogno di poter modificare il codice macchina di eventuali istruzioni del tipo:

```
mov ax, DATASEGM
```

E' chiaro infatti che, se **DATASEGM** viene rilocato dal **DOS**, e' anche necessario modificare il codice macchina della precedente istruzione; per poter svolgere questo lavoro, il **DOS** ha bisogno di conoscere il cosiddetto **relocation record** di ciascuna istruzione che fa' riferimento ad un segmento rilocabile. Il **relocation record** e' l'indirizzo **Seg:Offset** di un **Imm16** che, in una istruzione, rappresenta una componente **Seg** rilocabile; come esempio pratico, consideriamo proprio la precedente istruzione, alla quale corrisponde, in Figura 2, il codice macchina:

```
B8h 0000h
```

L'assembler ha assegnato a questo codice macchina, l'offset **0000h** calcolato rispetto all'inizio di **CODESEGM**; il linker ha assegnato a **DATASEGM** il valore **0000h**, a **CODESEGM** il valore **0044h**, e ha rilocato (definitivamente) l'offset della precedente istruzione, portandolo a **0008h**.
In seguito alle modifiche apportate dal linker, il codice macchina di questa istruzione rimane invariato, mentre il suo indirizzo logico iniziale diventa **0044h:0008h**; il valore che il **DOS** deve rilocare nel codice macchina, e' **0000h**, che si trova chiaramente all'indirizzo logico **0044h:0009h** (**1** byte dopo il codice **B8h**). Di conseguenza, il **relocation record** relativo a questa istruzione e' **0044h:0009h**.
Nel momento in cui **EXETEST.EXE** viene caricato in memoria, il **DOS** procede alla rilocazione dei segmenti **DATASEGM**, **CODESEGM** e **STACKSEGM** (ovviamente, vengono rilocate anche le componenti **Seg** dei vari **relocation records**); subito dopo, il **DOS** accede alle istruzioni puntate dai vari **relocation records**, e modifica ogni riferimento agli stessi **DATASEGM**, **CODESEGM** e **STACKSEGM**.

Tutte le informazioni di cui il **DOS** ha bisogno per poter gestire un file **EXE**, vengono generate dal linker; a tale proposito, come e' stato gia' anticipato, il linker suddivide un file **EXE** in due parti chiamate, **header** (intestazione), e **loadable module** (modulo caricabile, contenente il programma vero e proprio). All'interno dell'**header**, vengono infilate tutte le informazioni richieste dal **DOS**; la Figura 8 contiene la descrizione dell'**header** di un file **EXE** (per maggiori dettagli, si consiglia di consultare un manuale per programmatori **DOS**).

Header di un file EXE

Il formato **EXE** conferisce ai programmi **DOS** eseguibili, la massima flessibilita' ed efficienza; l'altro formato disponibile in ambiente **DOS**, e' il formato **COM** (COre iMage). La struttura di un eseguibile in formato **COM**, e' la piu' semplice possibile; nel caso degli **EXE** invece, e' possibile realizzare programmi aventi una struttura estremamente complessa.
Per poter garantire queste caratteristiche, i linker creano dei file **EXE** che iniziano con una intestazione (**header**) contenente una serie di informazioni destinate al sistema operativo (**DOS**); la parte successiva del file **EXE**, contiene il cosiddetto **modulo caricabile**, cioe' i blocchi di codice, dati e stack che, al momento dell'esecuzione, verranno caricati in memoria dal **SO**.

L'**header** e' formato da una parte di lunghezza fissa chiamata **struttura di controllo**, e da una parte di lunghezza variabile; in ogni caso, la dimensione dell'**header** e' un multiplo intero di **16** byte, e non deve mai essere inferiore a **512** byte.

Parte di lunghezza fissa dell'header

Analizziamo ora le caratteristiche, in versione **Assembly**, della struttura di controllo dell'**header**; per comodita', sulla sinistra della struttura sono stati riportati anche gli offset interni dei vari membri.

```
0000h    ExeFileControlData STRUC

0000h       Signature               db    'MZ'
0002h       LastPageSize            dw    ?
0004h       PageCount               dw    ?
0006h       RelocationRecords       dw    ?
0008h       HeaderParagraphs        dw    ?
000Ah       MinimumAllocation       dw    ?
000Ch       MaximumAllocation       dw    ?
000Eh       InitialSS               dw    ?
0010h       InitialSP               dw    ?
0012h       CheckSum                dw    ?
0014h       InitialIP               dw    ?
0016h       InitialCS               dw    ?
0018h       StartOfRelocationTable  dw    ?
001Ah       OverlayNumber           dw    ?

001Ch    ExeFileControlData ENDS
```

Analizziamo in dettaglio, il significato dei vari membri di questa importante struttura.

Signature (firma).
Questo campo contiene la stringa **'MZ'**, formata quindi dai due codici **ASCII 4Dh** e **5Ah**; questa stringa rappresenta le iniziali di **Mark Zbikowsky**, che e' uno dei programmatori che hanno lavorato alla definizione dello standard **EXE.** Al momento di caricare un eseguibile in memoria, se il **DOS** trova questa stringa, tratta il programma come un **EXE**; in caso contrario lo tratta come un **COM.**

LastPageSize (lunghezza dell'ultima pagina).
Per maneggiare comodamente un file **EXE**, il **DOS** lo suddivide in blocchi da **512** byte ciascuno, chiamati **pagine**; la dimensione dell'ultimo blocco, ovviamente, sara' sempre minore o uguale a **512** byte, e per questo motivo, e' necessario il campo **LastPageSize**, che indica appunto la dimensione in byte dell'ultima pagina.

PageCount (numero pagine).
Questo campo, contiene il numero complessivo delle pagine da **512** byte ciascuna, in cui il **DOS** suddivide i file **EXE**; tale numero comprende anche l'ultima pagina.

Di conseguenza, la dimensione in byte del file **EXE**, e' pari a:
((PageCount - 1) * 512) + LastPageSize.

RelocationRecords (numero elementi rilocabili).
Questo campo contiene il numero degli elementi che formano la cosiddetta **tabella di rilocazione**; le caratteristiche di questa tabella, vengono descritte piu' avanti.

HeaderParagraphs (dimensione dell'**header** in paragrafi).
Questo campo contiene la dimensione in paragrafi (blocchi da **16** byte) dell'**header** del file **EXE**; questo significa che la parte del file **EXE** che verra' caricata in memoria (modulo caricabile), si trova a (**HeaderParagraphs x 16**) byte di distanza dall'inizio del file **EXE** stesso.

MinimumAllocation (memoria minima allocata per il programma).
Questo campo indica il numero minimo di paragrafi di memoria aggiuntiva, che verranno allocati dal **SO** per garantire il corretto funzionamento del programma eseguibile; questi paragrafi aggiuntivi, vengono posti alla fine del blocco di memoria contenente il programma eseguibile, e il loro scopo e' quello di contenere dati non inizializzati del programma stesso (in particolare, lo stack).
Complessivamente, la memoria minima totale in byte, riservata all'intero programma, sara' pari quindi alla somma tra la dimensione in byte del modulo caricabile, e il valore (**MinimumAllocation x 16**).

MaximumAllocation (memoria massima allocata per il programma).
Questo campo indica il numero massimo di paragrafi di memoria aggiuntiva, che verranno allocati dal **SO** per il programma eseguibile; in genere, i linker inizializzano questa **WORD** a **FFFFh** (**65536** paragrafi, pari a **65536x16=1** Mb). Non essendo possibile trovare un blocco di memoria da **1** Mb, il **DOS** riserva ad ogni programma eseguibile, il piu' grande blocco di memoria che riesce a trovare; anche questi paragrafi aggiuntivi, vengono posti a partire dalla fine del blocco di memoria contenente il programma eseguibile.
Complessivamente, la memoria massima totale in byte che verra' occupata dal programma, sara' quindi pari alla somma tra la dimensione in byte del modulo caricabile, e il valore (**MaximumAllocation x 16**).

InitialSS (valore iniziale di **SS**).
In questo campo, il linker inserisce la componente **Seg** dell'indirizzo logico da cui inizia il blocco stack; si tratta chiaramente di un valore destinato ad essere rilocato dal **SO**.
Supponiamo che un programma venga caricato in memoria a partire dal segmento di memoria **StartSeg**; il **SO**, provvedera' allora a rilocare la componente **Seg** dell'indirizzo di partenza del blocco stack (**StackSeg**), ponendo:
StackSeg = InitialSS + StartSeg.
Il valore cosi' calcolato, viene caricato in **SS**.

InitialSP (valore iniziale di **SP**).
In questo campo, il linker inserisce l'offset iniziale che verra' caricato nel registro **SP** al momento dell'esecuzione del programma; trattandosi di un offset relativo a **SS**, non e' necessaria nessuna rilocazione.

CheckSum (valore di controllo).
Questo campo contiene un codice generato dal linker, che ha lo scopo di "validare" il file **EXE**; al momento di caricare il programma in memoria, il **SO** controlla questo valore per sapere se ha a che fare con un eseguibile valido.
Il codice contenuto in **CheckSum**, viene determinato dal linker in base ad un apposito algoritmo; nel caso degli eseguibili **DOS**, l'algoritmo consiste innanzi tutto nel suddividere il file **EXE** in blocchi da **16** bit (**WORD**). A questo punto viene calcolata la somma dei valori binari contenuti in tutte queste **WORD**; nel campo **CheckSum** viene messo il valore che porta questa somma a **FFFFh**.
Se il file **EXE** e' formato da un numero dispari di byte, il linker calcola il **CheckSum**, ponendo a zero il byte piu' significativo dell'ultima **WORD**. Prima di

caricare il programma in memoria, il **SO** ripete questo calcolo e lo confronta con il risultato ottenuto dal linker; se i due valori non coincidono, il file **EXE** viene considerato non valido (corrupted).
Come si puo' notare, si tratta di un sistema di validazione piuttosto banale, che non puo' certo proteggere un eseguibile dai virus; nelle versioni piu' recenti del **DOS**, questo campo viene ignorato.

InitialIP (valore iniziale di **IP**).
In questo campo, il linker inserisce l'offset dell'**entry point** del programma, che verra' utilizzato per inizializzare **IP**; anche in questo caso, trattandosi di un offset relativo a **CS**, non e' necessaria nessuna rilocazione.

InitialCS (valore iniziale di **CS**).
In questo campo, il linker inserisce la componente **Seg** dell'indirizzo logico da cui inizia il blocco codice che contiene l'**entry point** del programma; come accade per il blocco stack, anche questa componente **Seg** e' destinata ad essere rilocata dal **SO**.
In riferimento allo stesso esempio fatto per lo stack, se il programma viene caricato in memoria a partire dal segmento di memoria **StartSeg**, il **SO** esegue la rilocazione della componente **Seg** dell'indirizzo di partenza del blocco codice (**CodeSeg**) secondo la formula:
CodeSeg = InitialCS + StartSeg.
Il valore cosi' calcolato, viene caricato in **CS**.

StartOfRelocationTable (inizio della tabella di rilocazione).
Questo campo contiene l'offset, calcolato rispetto all'inizio del file **EXE**, da cui parte la tabella di rilocazione descritta piu' avanti.

OverlayNumber (numero di overlay).
Questo campo, viene utilizzato dai programmi che gestiscono le **overlay**; si tratta di un vecchio sistema che, in situazioni di scarsita' di memoria, consente ugualmente l'esecuzione di programmi, che richiedono piu' memoria di quella fisicamente presente sul computer. Attraverso questo sistema, il programma viene suddiviso in tante parti, chiamate appunto **overlay**; in fase di esecuzione, vengono caricate in memoria solo le parti necessarie del programma, mentre le parti non necessarie, rimangono sul disco. Come e' stato gia' detto, si tratta di un vecchio sistema che ormai non viene piu' utilizzato; per i programmi che non fanno uso di **overlay**, il valore di questo campo viene posto a **0000h**.

Parte di lunghezza variabile dell'header

La parte di lunghezza variabile dell'**header**, contiene informazioni relative alle **overlay** e ai **relocation records**.

Informazioni sulle overlay

Questa parte dell'**header**, contiene informazioni legate ai programmi che fanno uso delle **overlay**; quest'area e' riservata ai gestori di **overlay**, e le informazioni in essa contenute, non hanno niente a che vedere con il **SO**.

Tabella di rilocazione

In seguito alla rilocazione che il **DOS** effettua sulle componenti **Seg** degli indirizzi iniziali dei vari segmenti di programma, si rende necessaria anche la modifica del codice macchina di tutte le istruzioni che contengono riferimenti a queste stesse componenti **Seg**; la tabella di rilocazione, contiene un elenco di coppie **Seg:Offset** (chiamate **relocation records**), ciascuna delle quali si riferisce ad una componente **Seg** da rilocare.
La tabella di rilocazione si trova a **StartOfRelocationTable** byte di distanza,

dall'inizio del file **EXE**; il numero di **relocation records** di questa tabella, e' contenuto nel campo **RelocationRecords** della struttura di controllo. In versione **Assembly**, ogni **relocation record** della tabella di rilocazione, assume l'aspetto seguente:

```
RelocationRecord STRUC

  RelocationOffset     dw   ?
  RelocationSegment    dw   ?

RelocationRecord ENDS
```

Supponiamo ad esempio, che nel blocco codice **CODESEGM** di un programma, siano presenti **3** istruzioni che fanno riferimento ad un blocco dati chiamato **DATASEGM**; supponiamo anche che il linker, abbia assegnato a **DATASEGM** il paragrafo **002Bh**, e a **CODESEGM** il paragrafo **004Ch**. Nell'ipotesi che i tre riferimenti a **DATASEGM** si trovino agli offset **003Ah**, **01F2h** e **028Ch** di **CODESEGM**, otteniamo nella struttura di controllo:

RelocationRecords = 3

Inoltre, la **relocation table** assumera' la seguente struttura:

003Ah
004Ch

01F2h
004Ch

028Ch

004Ch

Se la dimensione dell'**header** e' inferiore a **512** byte, il linker aggiunge un numero opportuno di byte di valore **00h**, sino ad arrivare appunto a **512** byte; in ogni caso, la dimensione in byte dell'**header** deve essere sempre un multiplo intero di **16**. Subito dopo l'**header**, inizia il **modulo caricabile**, contenente il codice, i dati e lo stack del programma.

Attraverso il contenuto dell'**header**, possiamo ricavare numerose informazioni sul file **EXE**; per calcolare, ad esempio, la dimensione in byte del solo modulo caricabile, dobbiamo procedere in questo modo:

DimensioneFileEXE = ((PageCount - 1) * 512) + LastPageSize

DimensioneHeader = HeaderParagraphs * 16

DimensioneModuloCaricabile = DimensioneFileEXE – DimensioneHeader

Nel prossimo capitolo, analizzeremo alcuni metodi di disassemblaggio di un programma eseguibile; in questo modo, sara' possibile esaminare in pratica, l'**header** di un file **EXE**. La Figura 9, visualizza una serie di informazioni su **EXETEST.EXE**, facilmente deducibili anche senza consultare il contenuto dell'**header**.

Figura 9 - Informazioni su EXETEST.EXE

```
Struttura di controllo:

   InitialSS         = 0059h
   InitialSP         = 0400h
   InitialCS         = 0044h
   InitialIP         = 0008h
   RelocationRecords = 1

Tabella di rilocazione

   RelocationOffset  = 0009h
   RelocationSegment = 0044h
```

13.7 Esecuzione di un file EXE

Una volta che abbiamo a disposizione il nostro programma **EXETEST.EXE**, possiamo procedere con la sua esecuzione; a tale proposito, dobbiamo posizionarci nella cartella **ASMBASE**, e impartire il comando:

```
exetest
```

Premendo ora il tasto **[Invio]**, cediamo il controllo all'interprete dei comandi **COMMAND.COM**; l'interprete dei comandi, si accorge subito che **exetest** non e' un comando **DOS**, e si mette allora alla ricerca, nella cartella **ASMBASE**, di un eseguibile di nome **EXETEST**.
In base alle convenzioni seguite dal **DOS**, **COMMAND.COM** cerca un file chiamato **EXETEST.COM**; se questo file non viene trovato, **COMMAND.COM** cerca un file chiamato **EXETEST.EXE**. Se questo file non viene trovato, **COMMAND.COM** cerca un file chiamato **EXETEST.BAT**; i file con estensione **BAT** vengono chiamati **batch file**, e sono dei semplici file di testo contenenti una sequenza di comandi **DOS**. Se il file **EXETEST.BAT** non viene trovato, **COMMAND.COM** genera un messaggio di errore del tipo:

```
Comando o nome file non valido
```

Nel nostro caso, nella cartella **ASMBASE**, l'interprete dei comandi trova il file **EXETEST.EXE**; supponendo che si tratti di un file eseguibile, **COMMAND.COM** cede il controllo ad un apposito programma del **SO**, chiamato **loader** (caricatore). Come si intuisce dal nome, il compito del **loader** e' quello di caricare in memoria i programmi eseguibili; a tale proposito, il **loader** consulta l'eventuale **header** del file da eseguire. Se viene trovata la stringa iniziale **'MZ'**, il file viene trattato come **EXE**; in caso contrario, il **loader** cerca di trattare **EXETEST** come un eseguibile di tipo **COM**. E' chiaro che, se proviamo ad eseguire un "finto" file **EXE** o **COM**, provochiamo una situazione di errore che puo' anche determinare un crash (**Windows** visualizza una finestra di errore).
Nel caso di **EXETEST.EXE**, il **loader** trova la stringa **'MZ'**, e quindi, avvia le procedure standard per il caricamento in memoria di un programma **EXE**; per ogni programma, **EXE** o **COM**, da caricare in memoria, vengono predisposti due blocchi di memoria chiamati **environment segment** e **program segment**. Come e' stato spiegato in un precedente capitolo, tutti i blocchi di memoria forniti dal **DOS**, sono sempre allineati al paragrafo; inoltre, la dimensione in byte di un qualunque blocco di memoria **DOS**, e' sempre un multiplo intero di **16**.

13.7.1 L'environment segment

L'**environment segment** o **segmento d'ambiente** di un programma, e' un blocco di memoria **DOS** destinato a contenere una sequenza di stringhe, chiamate **variabili d'ambiente**; ogni stringa assume la forma:
`'NOME=contenuto', 0`
Come si puo' notare, queste stringhe seguono le convenzioni del linguaggio **C**; una stringa **C** e' un vettore di codici ASCII, terminato da un byte di valore **0** (il codice ASCII **0**, viene anche chiamato **NULL**).
Subito dopo l'ultima stringa della sequenza, e' presente un byte di valore **0**, che rappresenta quindi una stringa **C** vuota.
Nelle versioni piu' recenti del **DOS**, subito dopo questo byte, e' presente una word contenente il numero di eventuali stringhe extra; nel caso generale, e' presente una sola stringa extra che contiene il nome dell'eseguibile, completo di percorso (**path**).

Le variabili d'ambiente di un programma, vengono passate da chi ha richiesto l'esecuzione del programma stesso; nel caso piu' frequente, l'esecuzione di un programma viene richiesta da **COMMAND.COM**. Anche nell'esempio presentato in questo capitolo, e' proprio **COMMAND.COM** che ha richiesto l'esecuzione di **EXETEST.EXE**; si dice allora che **COMMAND.COM**, e' **padre** di **EXETEST.EXE**, oppure che **EXETEST.EXE**, e' **figlio** di **COMMAND.COM**. A sua volta, **EXETEST.EXE** potrebbe richiedere l'esecuzione di un altro programma, chiamato ad esempio, **TEST2.EXE**; in questo caso, **EXETEST.EXE**, come **padre** di **TEST2.EXE**, deve provvedere a passare le variabili d'ambiente allo stesso **TEST2.EXE**.
La Figura 10, mostra un esempio in versione **Assembly**, della struttura del segmento d'ambiente, che **COMMAND.COM** crea per il programma figlio **EXETEST.EXE**.

Figura 10 - Variabili d'ambiente di EXETEST.EXE

```
db    'COMSPEC=C:\COMMAND.COM', 0              ; variabile COMSPEC
db    'PATH=C:\DOS;C:\WINDOWS;C:\TASM\BIN', 0   ; variabile PATH
db    'PROMPT=$p$g', 0                          ; variabile PROMPT
db    0                                         ; stringa vuota
dw    1                                         ; numero stringhe extra
db    'C:\TASM\ASMBASE\EXETEST.EXE', 0          ; stringa extra
```

La variabile **COMSPEC** viene inizializzata dal **SO**, e viene usata dai programmi, per invocare l'interprete dei comandi **COMMAND.COM**; questo e' proprio quello che accade, quando si lancia la **DOS Box** dall'interno di **Windows98**.
Le due variabili **PATH** e **PROMPT** di Figura 10, vengono invece configurate dall'utente attraverso il file **AUTOEXEC.BAT**; a tale proposito, all'interno del file **C:\AUTOEXEC.BAT**, si puo' utilizzare il comando **SET** per scrivere ad esempio:
`SET PATH=C:\DOS;C:\WINDOWS;C:\TASM\BIN`
Per approfondire questi concetti, si consiglia di consultare un manuale per programmatori **DOS**.

Chi conosce il linguaggio **C**, sara' interessato a sapere che l'unica stringa extra presente nel segmento d'ambiente, rappresenta il famoso argomento **argv[0]** passato (dal compilatore **C**) alla funzione:
`int main(int argc, char *argv[])`

Un segmento d'ambiente, puo' arrivare a contenere sino a **32** Kb di informazioni; bisogna prestare quindi particolare attenzione al fatto che, numerose richieste di esecuzione a catena, da un programma ad un altro, possono occupare una notevole quantita' di memoria convenzionale.

13.7.2 Il program segment

Il **program segment** o **segmento di programma** (da non confondere con i segmenti che formano un programma), e' un blocco di memoria **DOS** destinato a contenere il **PSP** o **Program Segment Prefix** (prefisso del segmento di programma), e il programma eseguibile vero e proprio; la Figura 11 illustra le caratteristiche del **PSP** (per maggiori dettagli, si consiglia di consultare un manuale per programmatori **DOS**).

Figura 11 - Il Program Segment Prefix

Il **Program Segment Prefix** (abbreviato in **PSP**), occupa i primi **256** byte (**16** paragrafi) del **program segment** assegnato ad un programma **EXE** o **COM** da eseguire; siccome il **program segment** e' allineato al paragrafo, anche il **PSP** sara', a sua volta, allineato al paragrafo. Lo scopo del **PSP**, e' quello di contenere una serie di importanti informazioni, che vengono utilizzate, sia dal **DOS**, sia dal programma in esecuzione.
Analizziamo le caratteristiche, in versione **Assembly**, della struttura del **PSP**; per comodita', sulla sinistra della struttura sono stati riportati anche gli offset interni dei vari membri.

```
0000h    ProgramSegmentPrefix STRUC

0000h       TerminateLocation       dw    ?
0002h       EndOfAllocatedMemory    dw    ?
0004h                               db    ?
0005h       OldDosCallLocation      db    5   dup (?)
000Ah       SavedInt22Pointer       dd    ?
000Eh       SavedInt23Pointer       dd    ?
0012h       SavedInt24Pointer       dd    ?
0016h       ParentPspSegment        dw    ?
0018h       HandleIndexTable        db    20  dup (?)
002Ch       EnvironmentSegment      dw    ?
002Eh                               db    4   dup (?)
0032h       HandleIndexTableLength  dw    ?
0034h       HandleIndexTablePtr     dd    ?
0038h                               db    24  dup (?)
0050h       NewDosCallLocation      db    3   dup (?)
0053h                               db    9   dup (?)
005Ch       FirstFCB                db    16  dup (?)
006Ch       SecondFCB               db    16  dup (?)
007Ch                               db    4   dup (?)
0080h       CommandLineParmLength   db    ?
0081h       CommandLineParameters   db    127 dup (?)

0100h    ProgramSegmentPrefix ENDS
```

Il contenuto di diversi membri del **PSP**, rimane inutilizzato, in quanto si riferisce alle primissime versioni del **DOS**, che risentivano ancora dell'influenza del **SO** predecessore, cioe' il **CP/M** (**Control Program for Microcomputers**); come molti ricorderanno, il **CP/M**, sviluppato dalla **Digital**, veniva utilizzato anche da alcuni modelli di microcomputers **Commodore**.
Quando la **IBM** decise di mettere in commercio il primo personal computer (**XT**), si rivolse proprio alla **Digital** per la realizzazione del **SO**; in pratica, alla **Digital** venne chiesto di convertire a **16** bit il suo **SO CP/M** a **8** bit, in modo da adattarlo alla **CPU 8086** che equipaggiava l'**XT**. Per motivi legati, probabilmente, ai diritti d'autore, le trattative tra le due aziende vennero bruscamente interrotte; fu proprio a quel punto, che si fecero avanti due studenti che, con poco profitto, frequentavano l'universita' di **Harvard**. Quei due studenti, poco tempo prima avevano fondato una piccola software-house, chiamata all'epoca **Micro-Soft** (con il trattino); come molti avranno capito, quei due personaggi erano **Bill Gates** e **Paul Allen**. In modo a dir poco misterioso, il **CP/M** fini'

nelle mani di **Gates** e **Allen**, che lo "girarono" poi alla **IBM** in versione a **16 bit**; questo "nuovo" sistema operativo, prese il nome di **MS-DOS 1.0** (**Microsoft Disk Operating System**).

Dopo queste necessarie precisazioni, analizziamo il significato dei campi che formano il **PSP**; si tenga presente che, i campi privi di nome, sono riservati al **DOS**.

TerminateLocation (indirizzo di fine esecuzione).
Questo campo da **16** bit, contiene il codice macchina di una istruzione (**INT 20h**) che termina il programma in esecuzione, e restituisce il controllo al **DOS**; in sostanza, se un programma vuole terminare la propria esecuzione, deve effettuare un salto all'offset **0000h** del **PSP**. Si ottiene lo stesso risultato, attraverso una chiamata diretta alla **INT 20h**; questi metodi di terminazione di un programma, sono ormai obsoleti, e continuano ad esistere solo per motivi di compatibilita' con le vecchissime versioni del **DOS**.

EndOfAllocatedMemory (fine della memoria allocata).
Questo campo da **16** bit, contiene la componente **Seg** dell'indirizzo, da cui inizia il primo segmento di memoria, successivo al **program segment** del programma in esecuzione; possiamo dire quindi che, il primo segmento di memoria successivo al **program segment** del programma in esecuzione, parte dall'indirizzo logico **EndOfAllocatedMemory:0000h**.

OldDosCallLocation (vecchio indirizzo per le chiamate al **DOS**).
Questo campo da **5** byte, e' un retaggio del vecchio **CP/M**; i programmi che giravano su **CP/M**, richiedevano i servizi del **SO**, attraverso una chiamata al codice macchina presente appunto all'offset **0005h** del **PSP**. Naturalmente, questo campo non viene utilizzato dal **DOS**; nelle versioni piu' recenti del **DOS**, le richieste dei servizi del **SO**, vengono effettuate attraverso una chiamata della **INT 21h** (interrupt dei servizi **DOS**).

SavedInt22Pointer, **SavedInt23Pointer**, **SavedInt24Pointer** (copie degli indirizzi delle **ISR** per le **INT 22h**, **23h** e **24h**).
Prima di eseguire un programma, il **DOS** salva in questi tre campi da **32** bit ciascuno, gli indirizzi logici **Seg:Offset** delle **ISR** corrispondenti alle tre **INT**, **22h** (terminate address), **23h** (control break address) e **24h** (critical error address); in questo modo, il programma in esecuzione, puo' installare proprie **ISR** per gestire queste tre **INT**, senza preoccuparsi di salvare gli indirizzi delle **ISR** predefinite. Quando il programma termina, il **DOS** provvede ad effettuare tutte le operazioni di ripristino dei vecchi indirizzi.

ParentPspSegment (segmento **PSP** del padre).
Questo campo da **16** bit, contiene la componente **Seg** dell'indirizzo da cui inizia il **PSP** del **padre** del programma in esecuzione; siccome il **PSP** e' sempre allineato al paragrafo, possiamo dire che il **PSP** del **padre** del programma in esecuzione, inizia dall'indirizzo logico **ParentPspSegment:0000h**.

HandleIndexTable (tabella degli indici delle **handle**).
Un programma **DOS** in esecuzione, puo' aprire, se necessario, un certo numero di file su disco; quando si apre un file su disco, il **DOS** restituisce un valore a **16** bit, chiamato **handle** (maniglia), che viene utilizzato per identificare il file stesso. Le **handle** di tutti i file correntemente aperti da tutti i programmi in esecuzione, vengono memorizzate in una tabella globale che prende il nome di **SFT** o **System File Table**; ogni elemento di questa tabella, contiene la descrizione completa di uno dei file aperti.
Ogni programma in esecuzione, e' dotato a sua volta di una tabella locale chiamata **handle index table**; ogni elemento di questa tabella, e' un indice che individua nella **SFT** uno dei file aperti dal programma stesso.
Il campo **HandleIndexTable** e' un vettore di **20** byte, dove ciascun byte rappresenta un indice; questo sistema permette quindi ad un programma, di aprire sino a **20** file contemporaneamente.

EnvironmentSegment (segmento d'ambiente).
Questo campo da **16** bit, contiene la componente **Seg** dell'indirizzo da cui inizia in memoria l'**environment segment** di un programma in esecuzione; siccome l'**environment segment** e' sempre allineato al paragrafo, il suo indirizzo logico iniziale completo e' **EnvironmentSegment:0000h**. Un programma in esecuzione, pur potendo accedere al suo **environment segment**, non deve assolutamente modificarne il contenuto.

HandleIndexTableLength (lunghezza della tabella degli indici delle **handle**).
Questo campo da **16** bit, contiene il numero massimo degli elementi che possono trovare posto nella **handle index table**; se la tabella e' contenuta nel **PSP**, questo numero e' **20**.

HandleIndexTablePtr (indirizzo della tabella degli indici delle **handle**).
Se un programma vuole aprire piu' di **20** file contemporaneamente, deve disporre di una tabella piu' grande di quella contenuta nel **PSP**; il campo **HandleIndexTablePtr** da **32** bit, contiene l'indirizzo logico completo **Seg:Offset** da cui inizia in memoria la **handle index table**. Eventualmente, questo indirizzo puo' anche puntare alla stessa **handle index table** presente nel **PSP**.

NewDosCallLocation (nuovo indirizzo per le chiamate ai servizi **DOS**).
Questo campo da **3** byte, rappresenta una procedura che contiene il codice macchina delle due istruzioni:

INT 21h e RET

La prima istruzione chiama la **INT 21h** (interrupt dei servizi **DOS**); la seconda istruzione, restituisce il controllo al chiamante.
In sostanza, un programma che vuole eseguire questo codice, deve chiamare la procedura che si trova all'offset **0050h** del **PSP**; si tratta di un procedimento contorto, che puo' essere tranquillamente evitato chiamando la **INT 21h** direttamente dal programma.

FirstFCB, **SecondFCB** (prima e seconda **FCB**).
Gli **FCB** o **File Control Blocks**, rappresentavano il sistema utilizzato nella versione **1.0** del **DOS**, per la gestione dei file su disco; questo sistema e' ormai obsoleto, ed e' stato sostituito dalla **SFT** o **System File Table**.

CommandLineParmLength (lunghezza effettiva della stringa dei parametri passati dalla linea di comando).
Questo campo da **1** byte, contiene la lunghezza effettiva della stringa, che l'utente ha passato dalla linea di comando, ad un programma da eseguire.

CommandLineParameters (stringa dei parametri passati dalla linea di comando).
Questo campo da **127** byte, contiene la stringa che racchiude l'insieme dei parametri che l'utente ha passato dalla linea di comando, ad un programma eseguibile; questa stringa termina con un byte di valore **0Dh**, che e' il codice **ASCII** del **carriage return** (ritorno carrello), corrispondente al tasto **[Invio]**.
Consideriamo ad esempio un programma **NOMEPROG.EXE** che viene eseguito con il comando:

nomeprog mela pera limone ciliegia

In questo caso, il campo **CommandLineParameters** assume, in versione **Assembly**, il seguente aspetto:

CommandLineParameters db ' mela pera limone ciliegia', 0Dh

Si noti che anche lo spazio (codice **ASCII 20h**) presente tra **nomeprog** e **mela**, viene inserito nella stringa dei parametri; in questo esempio, il campo **CommandLineParmLength** contiene il valore **26**, che non comprende il byte finale **0Dh**.
Se si ha la necessita' di elaborare queste informazioni, si consiglia di farlo non appena inizia l'esecuzione di un programma; la cosa migliore da fare, consiste nel salvare da qualche parte, il contenuto di **CommandLineParmLength** e **CommandLineParameters**.

13.7.3 Caricamento in memoria di EXETEST.EXE

Supponiamo ora che il **DOS**, abbia creato l'**environment segment** e il **program segment** per **EXETEST.EXE**; supponiamo anche che il **program segment** parta dall'indirizzo fisico (sempre allineato al paragrafo) **024B0h**. A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **024Bh:0000h**; possiamo dire quindi che il **program segment** destinato a contenere il modulo caricabile di **EXETEST.EXE**, inizia dall'offset **0000h** del segmento di memoria n. **024Bh**.
Nei primi **256** byte (**0100h** byte) del **program segment**, viene sistemato il **PSP**; possiamo dire quindi che il **PSP**, occupa tutti gli indirizzi fisici del **program segment**, compresi tra **024B0h** e:
`024B0h + (0100h - 1h) = 025AFh`
In termini di indirizzi logici, il **PSP** viene inserito nel segmento di memoria n. **024Bh**, e occupa tutti gli offset di questo segmento, compresi tra **0000h** e:
`0000h + (0100h - 1h) = 00FFh`
In sostanza, il **PSP** e' compreso tra gli indirizzi logici **024Bh:0000h** e **024Bh:00FFh**.

L'inizio del **PSP** coincide con l'inizio del **program segment**; per convenzione, si utilizza allora la sigla **PSP**, per indicare la componente **Seg** dell'indirizzo logico iniziale (**024Bh:0000h**) dello stesso **program segment**. Possiamo dire quindi che:
`PSP = 024Bh`

Immediatamente dopo il **PSP**, viene sistemato il modulo caricabile prelevato dal file **EXETEST.EXE**; siccome il **PSP** termina all'indirizzo fisico **025AFh**, possiamo dire che il modulo caricabile, parte dall'indirizzo fisico **025B0h**, allineato al paragrafo. A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **025Bh:0000h**; la componente **Seg** di questo indirizzo logico, e' proprio quella che viene utilizzata dal **DOS**, per rilocare i segmenti di **EXETEST.EXE**.
Indicando simbolicamente questa componente **Seg** con il nome **StartSeg**, possiamo scrivere:
`StartSeg = 025Bh`
Come si puo' notare, **StartSeg** si trova **16** paragrafi piu' avanti di **PSP**; a questo punto, con l'aiuto dell'**header** di Figura 9, il **DOS** procede con la rilocazione dei segmenti di programma di **EXETEST.EXE**.

Il linker aveva assegnato a **DATASEGM** l'indirizzo fisico iniziale **00000h**; il **DOS** somma questo indirizzo a **025B0h** e ottiene:
`00000h + 025B0h = 025B0h`
A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **025Bh:0000h**; di conseguenza:
`DATASEGM = 025Bh = StartSeg`
Come si puo' notare, gli offset interni a **DATASEGM** non subiscono nessuna rilocazione, e continuano a partire da **0000h** (confermando i calcoli del linker); osserviamo anche che l'indirizzo fisico iniziale **025B0h** di **DATASEGM**, e' compatibile con l'attributo di allineamento **PARA**.

Il linker aveva assegnato a **CODESEGM** l'indirizzo fisico iniziale **00448h**; il **DOS** somma questo indirizzo a **025B0h** e ottiene:
`00448h + 025B0h = 029F8h`
A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **029Fh:0008h**; di conseguenza:
`CODESEGM = 029Fh = InitialCS + StartSeg`
Come si puo' notare, gli offset interni a **CODESEGM** non subiscono nessuna rilocazione, e continuano a partire da **0008h** (confermando i calcoli del linker); osserviamo anche che l'indirizzo fisico iniziale **029F8h** di **CODESEGM**, e' compatibile con l'attributo di allineamento **DWORD**.

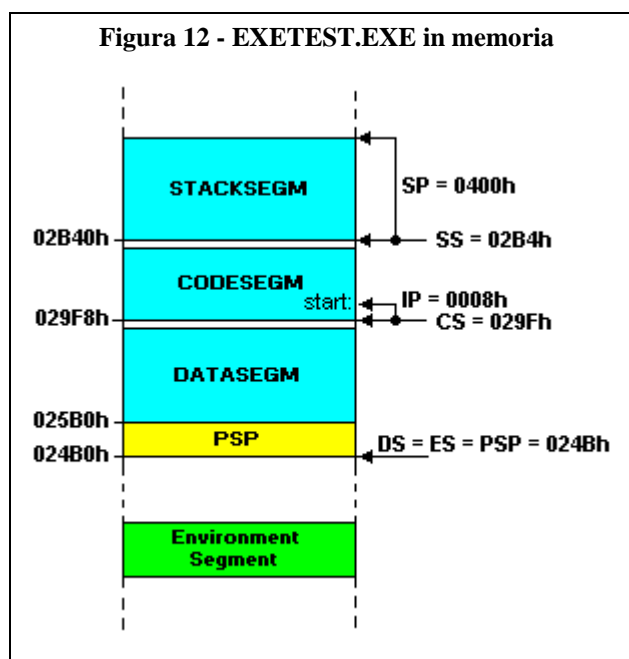
Il linker aveva assegnato a **STACKSEGM** l'indirizzo fisico iniziale **00590h**; il **DOS** somma questo indirizzo a **025B0h** e ottiene:

`00590h + 025B0h = 02B40h`
A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **02B4h:0000h**; di conseguenza:
`STACKSEGM = 02B4h = InitialSS + StartSeg`
Come si puo' notare, gli offset interni a **STACKSEGM** non subiscono nessuna rilocazione, e continuano a partire da **0000h** (confermando i calcoli del linker); osserviamo anche che l'indirizzo fisico iniziale **02B40h** di **STACKSEGM**, e' compatibile con l'attributo di allineamento **PARA**.

Nella **relocation table** (Figura 9), il **DOS** trova un solo **relocation record** rappresentato dalla coppia **0044h:0009h**; la componente **Seg** di questa coppia, viene ovviamente rilocata, e si ottiene:
`(0044h + StartSeg):0009h = 029Fh:0009h`
A questo indirizzo di **CODESEGM**, e' presente il valore da rilocare **0000h** (**DATASEGM**), relativo al codice macchina:
`B8h 0000h`
dell'istruzione:
`mov ax, DATASEGM`
Il valore **0000h** viene rilocato (sommato a **StartSeg**), e il precedente codice macchina diventa:
`B8h 025Bh`

Il **DOS** passa ora alla inizializzazione dei registri della **CPU**; per la coppia **CS:IP**, il **DOS** pone:
`CS:IP = CODESEGM:InitialIP = 029Fh:0008h`
Per la coppia **SS:SP**, il **DOS** pone:
`SS:SP = STACKSEGM:InitialSP = 02B4h:0400h`
Per i due registri di segmento **DS** e **ES**, il **DOS** pone per convenzione:
`DS = ES = PSP = 024Bh`

A questo punto, il programma e' pronto per l'esecuzione; in seguito a tutto questo lavoro svolto dal **DOS**, il programma **EXETEST.EXE** assume in memoria la disposizione illustrata in Figura 12.

Figura 12 - EXETEST.EXE in memoria

In un programma **EXE**, possono essere presenti, eventualmente, diversi segmenti di dati; proprio per questo motivo, il **SO** non puo' sapere quale di questi segmenti di dati, deve essere utilizzato per inizializzare **DS**. Questo compito quindi, viene lasciato al programmatore.

Convenzionalmente, il **DOS** inizializza **DS** e **ES**, con la componente **Seg** (che abbiamo chiamato **PSP**) dell'indirizzo iniziale del **program segment**; questo significa che se vogliamo accedere al contenuto del **PSP**, possiamo servirci di **DS** o **ES**, e delle informazioni visibili in Figura 11.
In genere, subito dopo l'**entry point** di un programma, **DS** viene inizializzato dal programmatore con un segmento di dati, per cui ci rimane a disposizione **ES**; in questo caso, il **PSP** e' accessibile a partire dall'indirizzo logico **ES:0000h**.
Il campo **CommandLineParmLength** ad esempio, si trova all'indirizzo logico **ES:0080h**; analogamente, il campo **CommandLineParameters**, si trova all'indirizzo logico **ES:0081h**.
L'**environment segment** del programma in esecuzione, si trova in memoria a partire dall'indirizzo logico **EnvironmentSegment:0000h**; a sua volta, il campo **EnvironmentSegment** si trova all'indirizzo logico **ES:002Ch**.
Molto spesso, si rende necessario anche il ricorso a **ES** per la gestione dei dati di un programma; proprio per questo motivo, l'elaborazione delle informazioni presenti nel **PSP**, deve essere effettuata prima che il contenuto iniziale di **DS** e **ES** venga sovrascritto.

13.7.4 Esecuzione di EXETEST.EXE

Una volta che il **DOS** ha inizializzato i necessari registri della **CPU**, puo' partire la fase di esecuzione di **EXETEST.EXE**; il controllo passa quindi alla stessa **CPU** che trova la coppia **CS:IP** inizializzata con l'**entry point 029Fh:0008h**. A questo indirizzo interno a **CODESEGM**, e' presente il codice macchina:

```
B8h 025Bh
```

La **CPU** quindi, decodifica ed esegue questa istruzione; nel frattempo, **IP** viene incrementato in modo che la coppia **CS:IP** punti alla prossima istruzione da eseguire.
La precedente istruzione, ha un codice macchina da **3** byte; di conseguenza, la **CPU** pone:

```
IP = IP + 0003h = 0008h + 0003h = 000Bh
```

All'indirizzo **029Fh:000Bh**, interno a **CODESEGM**, e' presente il codice macchina:

```
8Eh D8h
```

La **CPU** quindi, decodifica ed esegue questa istruzione; nel frattempo, **IP** viene incrementato in modo che la coppia **CS:IP** punti alla prossima istruzione da eseguire.
La precedente istruzione, ha un codice macchina da **2** byte; di conseguenza, la **CPU** pone:

```
IP = IP + 0002h = 000Bh + 0002h = 000Dh
```

e cosi' via.
L'esecuzione prosegue in questo modo, finche' non si arriva al blocco di istruzioni che terminano il programma e restituiscono il controllo al **DOS**.

13.8 La direttiva ALIGN

Il programma di esempio **EXETEST.EXE** presentato in questo capitolo, ha una finalita' puramente didattica, e non pone quindi nessuna attenzione, al problema del corretto allineamento delle informazioni in memoria; e' chiaro pero' che nei programmi reali, questo aspetto non puo' essere trascurato.

Come e' stato ampiamente detto nei precedenti capitoli, il programmatore **Assembly** ha la possibilita' di gestire in modo diretto, tutti i dettagli relativi al tipo di allineamento da assegnare, sia ai segmenti di programma, sia alle informazioni contenute nei segmenti stessi; codice e dati correttamente allineati in memoria, vengono acceduti alla massima velocita' possibile dalla **CPU**.
Nel capitolo precedente, abbiamo visto che grazie all'attributo **Allineamento**, possiamo richiedere al linker che un determinato segmento di programma, venga disposto in memoria a partire da un indirizzo fisico avente particolari requisiti di allineamento; questo attributo agisce quindi esclusivamente sull'indirizzo fisico iniziale dei segmenti di programma.

Se vogliamo imporre anche l'allineamento delle informazioni contenute all'interno di un segmento di programma, dobbiamo servirci della direttiva:
`ALIGN n`
Questa direttiva dice all'assembler che la prossima informazione, deve partire da un offset multiplo intero di **n**; il valore che possiamo assegnare ad **n** deve essere una potenza intera del **2**; e' possibile quindi allineare le informazioni ad offset multipli interi di **2**, **4**, **16**, **32**, etc. Per motivi abbastanza ovvi, il valore di **n** non puo' superare quello dell'attributo **Allineamento**; nel caso, ad esempio, di un segmento con attributo **PARA** (**16** byte), il valore di **n** non puo' superare **16**.
Naturalmente, nel caso di **CPU** con architettura a **8** bit, tutto cio' che riguarda l'allineamento dei segmenti di programma e del loro contenuto, non ha nessun significato; il discorso cambia invece, nel momento in cui abbiamo a che fare con **CPU** a **16** bit, **32** bit, **64** bit, etc.
Appare ovvio il fatto che la direttiva **ALIGN**, raggiunge la massima efficacia quando lavora in combinazione con l'attributo **Allineamento**; allineando, ad esempio, un segmento dati ad un indirizzo fisico pari, possiamo facilmente allineare ad indirizzi pari anche i dati contenuti nel segmento stesso. Per capire meglio questi importanti concetti, analizziamo in dettaglio i metodi da utilizzare per allineare in memoria, il codice, i dati statici e i dati temporanei del programma **EXETEST.EXE** presentato in questo capitolo.

13.8.1 Allineamento in memoria dei dati statici

Osservando il file **EXETEST.LST** di Figura 2, ci accorgiamo che nel segmento **DATASEGM**, l'allineamento dei dati appare piuttosto insoddisfacente; si nota infatti che il dato **varByte**, si trova ad un offset pari (**0000h**), mentre gli **11** dati successivi, si trovano tutti ad offset dispari.
Questa situazione, e' dovuta al fatto che **varByte**, che parte appunto da un offset pari, ha una ampiezza dispari (**1**) in byte, mentre gli **11** dati successivi, hanno tutti una ampiezza pari in byte; di conseguenza, **varByte** costringe gli **11** dati successivi, a posizionarsi tutti ad un offset dispari.
Se stiamo lavorando con **CPU** a **16** bit o superiori, si tratta di una situazione veramente pessima; infatti, questa situazione provoca un enorme aumento del numero di accessi in memoria, necessari alla **CPU** per leggere o scrivere i dati presenti in **DATASEGM**.

Per evitare questo problema, possiamo servirci di una semplice regola generale, valida per una qualsiasi **CPU** avente un **Data Bus** con ampiezza pari a **n** byte; questa regola consiste nell'inserire una direttiva **ALIGN n**, prima di qualsiasi dato il cui allineamento non e' un multiplo intero di **n**, e dopo qualsiasi dato la cui ampiezza in byte non e' un multiplo intero di **n**.

Per dimostrare questa regola, consideriamo una **CPU** con architettura a **16** bit, cioe' con **Data Bus** avente ampiezza pari a **2** byte; osservando il **listing file** di Figura 2, possiamo notare che, ad eccezione di **varByte** che ha allineamento pari e dimensione dispari, tutti gli altri **11** dati hanno allineamento dispari e ampiezza pari. Proviamo allora ad apportare una semplicissima modifica che consiste nell'inserire un buco di memoria da **1** byte tra **varByte** e **varWord**; la porzione di **DATASEGM** interessata da questa modifica, viene mostrata in Figura 13.

Figura 13 - Allineamento alla WORD dei dati di DATASEGM

```
DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

varByte     db          0bh                        ; intero a 1 byte
            db          ?                          ; buco da 1 byte
varWord     dw          0abcdh                     ; intero a 2 byte

..............................

DATASEGM    ENDS
```

Osserviamo subito che **varByte** continua a partire dall'offset **0000h**; subito dopo **varByte**, troviamo un buco di memoria da **1** byte che si trova all'offset **0001h**. Questo buco di memoria, fa' scalare in avanti di **1** byte gli offset di tutti gli altri **11** dati; inoltre, questi **11** dati hanno tutti una ampiezza pari in byte.
Riassemblando ora il programma di Figura 1, e consultando il nuovo **listing file**, possiamo constatare che con questa semplice modifica, tutti i dati presenti in **DATASEGM** si vengono a trovare allineati ad indirizzi pari.
Possiamo ottenere lo stesso risultato, attraverso la direttiva **ALIGN 2**; in questo modo si perviene alla situazione illustrata in Figura 14.

Figura 14 - Allineamento alla WORD dei dati di DATASEGM

```
DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

varByte     db          0bh                         ; intero a 1 byte
   ALIGN    2                                       ; allinea alla WORD
varWord     dw          0abcdh                      ; intero a 2 byte

.............................

DATASEGM    ENDS
```

Quando l'assembler esamina **DATASEGM**, assegna a **varByte** l'offset **0000h**; subito dopo, l'assembler incontra la direttiva **ALIGN**, attraverso la quale stiamo richiedendo un allineamento ad un offset pari per **varWord**. L'offset pari successivo a **0000h** e' ovviamente **0002h**; di conseguenza, l'assembler inserisce un buco di memoria da **1** byte subito dopo **varByte**, e fa' cosi' scalare in avanti di **1** byte gli offset di tutti i successivi **11** dati. Questi **11** dati hanno tutti una ampiezza pari in byte, per cui si troveranno tutti allineati ad un offset pari; alla fine si ottiene quindi la stessa identica situazione mostrata in Figura 13.

Veniamo ora al caso molto piu' frequente di una **CPU** con architettura a **32** bit, cioe' con **Data Bus** avente ampiezza pari a **4** byte; osservando il **listing file** di Figura 2, possiamo notare che questa volta, non basta inserire una direttiva **ALIGN 4** tra **varByte** e **varWord**. Questa direttiva, fa' scalare in avanti di **3** byte l'offset di **varWord**, che si viene cosi' a trovare all'offset **0004h**; a causa pero' dell'ampiezza pari a **2** byte della stessa **varWord**, la successiva variabile **varDword** si viene a trovare all'offset **0006h** che non e' un multiplo intero di **4**.
E' necessaria quindi una nuova direttiva **ALIGN 4** tra **varWord** e **varDword**; in questo modo, **varDword**, **varQword** e **varTbyte** (grazie alla loro ampiezza multipla intera di **4**), si vengono a trovare ad offset multipli interi di **4**.
La variabile **varTbyte** ha una ampiezza pari a **10** byte, per cui provoca il disallineamento di **varFloat32**; e' necessaria quindi una nuova direttiva **ALIGN 4** tra **varTbyte** e **varFloat32**, in modo che, **varFloat32**, **varFloat64**, **dataViaggio**, **varRect**, **vettRect** e **FiatPanda** (grazie alla loro ampiezza multipla intera di **4**), si vengano a trovare ad offset multipli interi di **4**.

La variabile **FiatPanda** ha una ampiezza pari a **46** byte, per cui provoca il disallineamento di **vettAuto**; e' necessaria quindi un'ultima direttiva **ALIGN 4** tra **FiatPanda** e **vettAuto**. Si perviene in questo modo alla situazione finale mostrata in Figura 15.

Figura 15 - Allineamento alla DWORD dei dati di DATASEGM

```
DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

varByte     db          0bh                     ; intero a 1 byte
   ALIGN    4                                   ; allinea alla DWORD
varWord     dw          0abcdh                  ; intero a 2 byte
   ALIGN    4                                   ; allinea alla DWORD
varDword    dd          12345678h               ; intero a 4 byte
varQword    dq          123456789abcdef0h       ; intero a 6 byte
varTbyte    dt          00001111222233334444h   ; intero a 10 byte
   ALIGN    4                                   ; allinea alla DWORD
varFloat32  dd          3.14                    ; reale a 4 byte
varFloat64  dq          -5.12345E-200           ; reale a 8 byte

DataViaggio RecData     < 12, 06, 2001 >        ; RecData a 4 byte
varRect     Rect        < >                     ; Rect a 8 byte
vettRect    Rect        10 dup ( < > )          ; vettore di 10 Rect
FiatPanda   Automobile  < >                     ; Automobile a 46 byte
   ALIGN    4                                   ; allinea alla DWORD
vettAuto    Automobile  20 dup ( < > )          ; vettore di 20 Automobile

DATASEGM    ENDS
```

Se vogliamo svolgere un lavoro piu' sofisticato, possiamo ricordare che con le **CPU** a **32** bit, i dati di tipo **WORD** non hanno problemi di allineamento, purche' si trovino ad indirizzi pari; sfruttando questo aspetto, possiamo inserire una direttiva **ALIGN 2** tra **varByte** e **varWord**. Questa modifica ci permette di eliminare la direttiva **ALIGN 4** tra **varWord** e **varDword**; in questo modo inoltre, riduciamo anche le dimensioni complessive di **DATASEGM**, in quanto le direttive **ALIGN**, comportano un certo spreco di memoria.

Nota importante.

Qualcuno obiettera' che in caso di rilocazione degli indirizzi da parte del linker, tutto il lavoro di allineamento svolto sul contenuto di **DATASEGM**, potrebbe rivelarsi inutile; in realta', questo pericolo esiste solo se assegnamo allo stesso **DATASEGM**, un attributo di allineamento generico di tipo **BYTE** (nessun allineamento)!

E' importante quindi ribadire che la direttiva **ALIGN**, deve lavorare in combinazione con l'attributo di allineamento del segmento di programma che contiene la direttiva stessa; nel caso di Figura 15 ad esempio, possiamo constatare che le varie direttive **ALIGN 4** diventano efficaci solo se **DATASEGM** viene fatto partire da un indirizzo fisico multiplo intero di **4** (attributo **DWORD** o **PARA**).

In presenza di un attributo **DWORD** ad esempio, il linker fara' partire **DATASEGM**, sempre da un indirizzo fisico multiplo intero di **4**; di conseguenza, tutte le informazioni contenute in questo segmento, resteranno allineate alla **DWORD** anche dopo il caricamento in memoria del programma.

La Figura 15 ci permette anche di osservare che dati come **FiatPanda**, a causa della loro ampiezza in byte, creano problemi di allineamento ai dati successivi; possiamo dire allora che lavorando con una **CPU** avente **Data Bus** a **n** byte, sarebbe meglio definire dati statici con ampiezza multipla intera di **n** byte.
Per risolvere questo problema, possiamo inserire ad esempio opportuni buchi di memoria all'interno delle dichiarazioni dei dati; possiamo anche sfruttare il fatto che l'assembler ci permette di inserire le direttive **ALIGN** all'interno delle dichiarazioni di **STRUC** e **UNION**.
Supponiamo ad esempio di voler ottimizzare la struttura **Automobile** per le **CPU** a **32** bit; possiamo apportare allora le modifiche mostrate in Figura 16.

Figura 16 - Struttura Automobile

```
Automobile  STRUC

   CodModello  dw       ?
   Cilindrata  dw       ?
   Cavalli     dw       ?
      ALIGN    4
   Revisioni   RecData  10 DUP ( < > )

Automobile  ENDS
```

Ricordiamo ancora una volta, che tutte le **WORD** con allineamento pari, comportano un unico accesso in memoria da parte delle **CPU** con architettura a **32** bit; dalla Figura 16 si ricava allora quanto segue:

CodModello si trova all'offset **0000h** di **Automobile**
Cilindrata si trova all'offset **0002h** di **Automobile**
Cavalli si trova all'offset **0004h** di **Automobile**
Revisioni si trova all'offset **0008h** di **Automobile**

Allineando allora un dato di tipo **Automobile** ad un indirizzo fisico multiplo intero di **4**, ottimizziamo al massimo l'accesso al dato stesso da parte di una **CPU** a **32** bit; osserviamo inoltre che, grazie alla direttiva **ALIGN**, la struttura di Figura 16 assume una ampiezza in byte pari a **48** (multipla intera di **4**), che tiene allineato alla **DWORD** anche il dato successivo.

13.8.2 Allineamento in memoria dei dati temporanei

Come e' stato gia' detto nel capitolo 10, un programma in esecuzione puo' fare un uso veramente massiccio dello stack; inoltre, lo stack di un programma viene pesantemente utilizzato anche dalle **ISR** chiamate dalla **CPU** in risposta alle richieste di interruzione software e hardware. Il programmatore, e' tenuto quindi a curare con grande attenzione, il corretto allineamento delle informazioni presenti nello stack; vediamo allora come ci si deve comportare nel caso delle **CPU** con architettura a **16** e a **32** bit.

Nei precedenti capitoli, e' stato detto che le **CPU** con architettura a **16** bit, per la gestione dello stack forniscono le due istruzioni **PUSH** e **POP**, che accettano esclusivamente operandi di tipo **WORD**; in questo caso, il corretto allineamento dei dati temporanei, puo' essere ottenuto in modo estremamente semplice.
In base alle considerazioni svolte in precedenza (in relazione al blocco **DATASEGM**), dobbiamo innanzi tutto far partire il segmento di stack da un indirizzo fisico multiplo intero di **2**; e' necessario quindi evitare l'uso dell'attributo di allineamento **BYTE**. A questo punto, dobbiamo assegnare a **SP** un offset iniziale multiplo intero di **2**; con questi semplici accorgimenti, tutte le **WORD** inserite

nello stack si verranno a trovare ad indirizzi fisici pari, e verranno accedute alla massima velocita' possibile dalla **CPU**.

Nel caso delle **CPU** con architettura a **32** bit, la situazione e' piu' delicata; come gia' sappiamo, queste particolari **CPU** permettono di utilizzare le istruzioni **PUSH** e **POP**, con operandi, sia di tipo **WORD**, sia di tipo **DWORD**. La cosa migliore da fare, consiste allora nell'utilizzare **PUSH** e **POP**, esclusivamente con operandi di tipo **DWORD**; questa tecnica viene largamente utilizzata nei **SO** a **32** bit come **Windows** e **Linux**.
Ripetendo quindi il ragionamento precedente, dobbiamo innanzi tutto allineare il segmento di stack ad un indirizzo fisico multiplo intero di **4** (attributo **DWORD** o **PARA**); successivamente, dobbiamo assegnare a **SP** un offset iniziale multiplo intero di **4**. In questo modo, tutte le **DWORD** presenti nello stack, si vengono a trovare ad indirizzi fisici allineati alla **DWORD**.

Se vogliamo usare **PUSH** e **POP** con operandi di tipo **WORD** e **DWORD**, la situazione diventa piu' complicata; in questo caso infatti, dobbiamo fare in modo che gli operandi di tipo **WORD** si trovino sempre ad indirizzi multipli interi di **2**, e che gli operandi di tipo **DWORD** si trovino sempre ad indirizzi multipli interi di **4**. Una tale situazione e' pero' troppo difficile da gestire; in casi del genere, si puo' anche tollerare un piccolo aumento di accessi in memoria, dovuto a qualche dato di tipo **DWORD** allineato ad un indirizzo che, pur essendo pari, non e' un multiplo intero di **4**.

13.8.3 Allineamento in memoria delle istruzioni

Come sappiamo, nelle **CPU** di classe **CISC**, vengono utilizzate delle istruzioni codificate con un numero variabile (e spesso anche dispari) di byte; cio' significa che in un segmento di codice, moltissime istruzioni si vengono a trovare fuori allineamento. E' chiaro pero' che sarebbe assurdo pensare di allineare ogni singola istruzione di un blocco codice; in questo modo infatti, si verificherebbe un enorme aumento delle dimensioni del programma.
Proprio per questo motivo, la strada che si segue consiste nell'allineare solamente quelle istruzioni che si trovano in posizioni particolarmente critiche; nel caso piu' frequente, si tratta di quelle istruzioni che vengono raggiunte dalla **CPU** attraverso un salto (con conseguente svuotamento della **prefetch queue**). Questi aspetti sono stati gia' anticipati nel capitolo 10 (sezione **10.6**); nei capitoli successivi, verranno analizzati diversi accorgimenti che permettono di ottenere un certo miglioramento nella velocita' di esecuzione del codice.

Quando l'assembler incontra una direttiva **ALIGN** in un segmento di dati, inserisce eventualmente dei buchi di memoria, rappresentati da un opportuno numero di byte, ciascuno dei quali vale **00h**; questo sistema, non puo' essere pero' utilizzato nei segmenti di codice, in quanto la **CPU** scambierebbe il valore **00h** per un codice macchina da eseguire (il codice **00h** indica infatti l'**Opcode** di una addizione tra sorgente **Reg8/Mem8** e destinazione **Mem8/Reg8**).
Per evitare questo problema, nei segmenti di codice l'assembler inserisce dei buchi di memoria, rappresentati da byte di valore **90h**; questo valore da **1** byte, e' il codice macchina completo dell'istruzione **NOP**. Il mnemonico **NOP** significa **no operation** (nessuna operazione da svolgere); quando la **CPU** incontra il codice macchina **90h**, incrementa di **1** il contenuto del registro **IP**, e passa quindi all'elaborazione dell'istruzione successiva.
Nel caso in cui sia necessario inserire buchi di memoria da piu' di **1** byte, l'assembler spesso si serve anche di istruzioni del tipo:

```
MOV AX, AX
```

Come si puo' facilmente constatare, questa istruzione lascia invariato il contenuto di **AX**; il suo unico scopo, e' quello di occupare memoria (in questo caso, il codice macchina **8Bh C0h**, occupa **2** byte).

13.9 Gli operatori SEG e OFFSET

Sulla base di cio' che e' stato esposto in questo capitolo in relazione alla rilocazione degli indirizzi da parte del linker e del **SO**, si intuisce subito il fatto che sarebbe assolutamente folle l'idea di scrivere programmi nei quali si tenta di determinare empiricamente le componenti **Seg** e **Offset** dell'indirizzo logico di una qualsiasi informazione; e' chiaro infatti, che un qualunque indirizzo logico **Seg:Offset** che nel codice sorgente assume un determinato aspetto, e' destinato a subire profonde modifiche ad opera del linker e del **SO**.
E' necessario quindi ribadire che il programmatore, deve evitare nella maniera piu' assoluta di scrivere programmi che tentano di calcolare indirizzi in modo empirico; per permettere al programmatore di svolgere questi calcoli in assoluta sicurezza, l'**Assembly** mette a disposizione due potenti operatori chiamati, **SEG** e **OFFSET**.

In riferimento al programma di Figura 1, consideriamo la seguente istruzione:
```
MOV BX, OFFSET varDword
```
Questa istruzione, trasferisce in **BX** la componente **Offset** dell'indirizzo logico del dato **varDword**; per capire bene questa istruzione, e' necessario ricordare che quando un dato viene caricato in memoria, il suo indirizzo logico **Seg:Offset** viene assegnato in modo definitivo (statico), e rimane fisso per tutta la fase di esecuzione. Tutto cio' significa che le componenti **Seg** e **Offset** dell'indirizzo logico di una informazione, non sono altro che due semplici valori immediati del tipo **Imm16**; ne consegue che la precedente istruzione, rappresenta un trasferimento dati da **Imm16** a **Reg16**.
Come gia' sappiamo, il codice macchina di questa istruzione e' formato dall'**Opcode 1011_w_Reg**, seguito dall'**Imm16**; nel nostro caso, **w=1** (operandi a **16** bit) e **Reg=BX=011b**. Otteniamo allora:
```
Opcode = 10111011b = BBh
```
Dal **listing file** di Figura 2, si rileva che l'offset di **varDword** e':
```
Disp16 = 0003h = 0000000000000011b
```
In definitiva, otteniamo il codice macchina:
```
10111011b 0000000000000011b = BBh 0003h
```
L'assembler, "marca" **0003h** come un offset rilocabile; in questo modo, il linker potra' apportare tutte le necessarie modifiche al precedente codice macchina. In base all'esempio su **EXETEST.EXE** presentato in questo capitolo, abbiamo visto che l'indirizzo logico che l'assembler assegna a **varDword** e' **0000h:0003h**; questo indirizzo logico viene trasformato dal linker in **0000h:0003h**, e dal **SO** in **025Bh:0003h**. In fase di esecuzione del programma, la componente **Offset** assegnata a **varDword** diventa definitiva (**0003h**), e il programmatore la puo' quindi utilizzare in modo sicuro.

Sempre in riferimento al programma di Figura 1, consideriamo la seguente istruzione:
```
MOV DX, SEG start
```
Questa istruzione, trasferisce in **DX** la componente **Seg** dell'indirizzo logico dell'etichetta **start**; anche in questo caso, abbiamo a che fare con un trasferimento dati da **Imm16** a **Reg16**. L'**Opcode** e' sempre **1011_w_Reg**; nel nostro caso, **w=1** (operandi a **16** bit) e **Reg=DX=010b**. Otteniamo allora:
```
Opcode = 10111010b = BAh
```
Dal **listing file** di Figura 2, si rileva che **start** si trova nel blocco **CODESEGM**, a cui l'assembler assegna il valore simbolico **0000h** (assolutamente privo di significato); in definitiva, otteniamo il codice macchina:
```
10111010b 0000000000000000b = BAh 0000h
```
L'assembler, "marca" **0000h** come un segmento rilocabile; in questo modo, il linker potra' apportare tutte le necessarie modifiche al precedente codice macchina. In base all'esempio su **EXETEST.EXE** presentato in questo capitolo, abbiamo visto che l'indirizzo logico che l'assembler assegna a **start** e' **0000h:0000h**; questo indirizzo logico viene trasformato dal linker in **0044h:0008h**, e dal **SO** in **029Fh:0008h**. In fase di esecuzione del programma, la componente **Seg**

assegnata a **start** diventa definitiva (**029Fh**), e il programmatore la puo' quindi utilizzare in modo sicuro.

Gli operatori **SEG** e **OFFSET** non vengono assolutamente influenzati da eventuali direttive **ASSUME**; e' chiaro infatti che l'assembler, incontrando l'istruzione:

```
MOV BX, OFFSET varDword
```

capisce benissimo che gli stiamo chiedendo di calcolare lo spiazzamento di **varDword** all'interno del segmento di appartenenza, che e' ovviamente **DATASEGM**. Per scrivere questa istruzione quindi, non e' necessaria una direttiva **ASSUME** che associa **DATASEGM** ad un qualsiasi registro di segmento.

Un altro aspetto importante da capire, riguarda il fatto che gli operatori **SEG** e **OFFSET**, lavorano solamente in fase di assemblaggio di un programma (**assembler time operators**); il loro scopo, e' quello di generare dei valori immediati (rilocabili), che vengono poi inseriti dall'assembler, direttamente nel codice macchina.

Nota importante.
In relazione ad un identificatore definito in un segmento di programma con attributo **USE16**, l'operatore **OFFSET** restituisce un valore di tipo **Imm16**; in relazione, invece, ad un identificatore definito in un segmento di programma con attributo **USE32**, l'operatore **OFFSET** restituisce un valore di tipo **Imm32**. E' fondamentale quindi ribadire che, in modalita' reale, bisogna definire segmenti di programma dotati di attributo **USE16**; in caso contrario, si va' incontro ad un sicuro crash!

13.10 Programmi eseguibili in formato COM

Come gia' sappiamo, il **SO CP/M** puo' essere definito l'antenato del **DOS**; ai tempi del **CP/M**, i computers disponevano di quantita' piuttosto limitate di memoria, in genere **32** Kb, **64** Kb o, al massimo, **128** Kb come nel caso del celebre **Commodore 128** (che utilizzava proprio questo **SO**).
In una situazione del genere, si rendeva necessario limitare al massimo le esigenze di memoria dei programmi; proprio per questo motivo, il **CP/M** prevedeva un formato eseguibile, che permetteva di realizzare programmi piuttosto piccoli e con ridottissime esigenze di memoria, a scapito ovviamente della flessibilita' del programma stesso.
Con l'arrivo del **DOS**, si rese necessaria la conversione verso questo nuovo **SO**, della grande quantita' di software scritto per il **CP/M**; per facilitare questa conversione, il **DOS** mise a disposizione un apposito formato per i programmi eseguibili, chiamato **COM** (COre Memory). Il formato **COM**, e' disponibile ancora oggi in ambiente **DOS**, e pur essendo stato soppiantato quasi del tutto dal formato **EXE**, viene ancora utilizzato per scrivere piccoli programmi, molto compatti ed efficienti.

13.10.1 Struttura di un eseguibile in formato COM

In pratica, un programma **COM** viene ottenuto riducendo all'essenziale la struttura di un programma **EXE**; per raggiungere questo obiettivo, si elimina innanzi tutto la possibilita' di definire piu' di un segmento di programma. In un programma **COM** quindi, e' presente un unico segmento, all'interno del quale trovano posto, il codice, i dati e lo stack; in presenza di una struttura cosi' semplice, molte delle informazioni che il linker inserisce nell'**header**, diventano superflue.
Spingendo al massimo la semplificazione, si arriva ad eliminare del tutto la necessita', da parte del linker, di generare l'**header**; di conseguenza, i file **COM** generati dal linker, contengono al loro interno solo ed esclusivamente il modulo caricabile.

In assenza dell'**header**, tutte le fasi di caricamento in memoria e inizializzazione di un programma **COM**, vengono gestite dal **SO** attraverso un procedimento standard; questo procedimento quindi, e' identico per qualsiasi file **COM**.

Nello scrivere un programma **Assembly** in formato **COM**, il programmatore ha l'obbligo di lasciare liberi i primi **256** byte (**0100h** byte) dell'unico segmento presente; come si puo' facilmente intuire, in questi primi **256** byte, il **SO** inserisce il **PSP**. Ne consegue che l'**entry point** di un programma **COM**, deve trovarsi obbligatoriamente all'offset **0100h** dell'unico segmento presente.

13.10.2 La direttiva ORG

Per liberare un certo numero di byte in una determinata zona di un segmento di programma, possiamo servirci di diversi metodi, piu' o meno complessi; il metodo piu' semplice ed efficace, consiste nel servirsi della direttiva:

```
ORG n
```

Quando l'assembler incontra questa direttiva mentre sta assemblando un segmento di programma, assegna il valore **n** alla componente **Offset** del **location counter** (**$**); l'argomento **n**, deve essere quindi un valore immediato di tipo **Imm16**.
Supponiamo ad esempio, che all'offset **0052h** di un segmento di programma chiamato **DATASEGM**, sia presente una direttiva:

```
ORG 0100h
```

In quel punto, **$** vale **DATASEGM:0052h**; in seguito a questa direttiva, l'assembler assegna a **$** il valore **DATASEGM:0100h**. Tutto lo spazio compreso tra l'offset **0052h** e l'offset **0100h** (estremi esclusi), viene riempito con dei buchi di memoria; l'assemblaggio di **DATASEGM** riprende quindi a partire dall'indirizzo logico **DATASEGM:0100h**.

In generale, l'argomento **n** specificato da **ORG**, puo' essere una qualunque espressione, formata pero' da operandi immediati; in riferimento al programma di Figura 1, in un punto del blocco **CODESEGM** possiamo scrivere ad esempio:

```
ORG STACK_SIZE - ((OFFSET start) + 00B2h)
```

L'assembler, deve avere la possibilita' di risolvere questa espressione, in modo da ottenere un valore finale di tipo **Imm16**.

13.10.3 Modello di programma Assembly in formato COM

Prima di illustrare la struttura generale di un programma **Assembly** in formato **COM**, rimane ancora da chiarire un aspetto importante, relativo al punto in cui inserire le definizioni dei dati statici; in presenza infatti di un unico segmento, c'e' il rischio che la **CPU** possa finire per sbaglio nel bel mezzo di questi dati, scambiandoli per codici macchina da eseguire!
Per risolvere questo problema, e' necessario sapere che a differenza di quanto accade con i programmi in formato **EXE**, in un programma in formato **COM** e' proibito inserire qualsiasi informazione prima dell'**entry point** (cioe' prima dell'offset **0100h**); vediamo allora quali altre strade si possono seguire, per poter individuare un'area adatta a contenere la definizione dei dati statici di un programma in formato **COM**.
Una prima soluzione, consiste nel definire tutti i dati statici, subito dopo l'**entry point**; in questo caso, prima delle varie definizioni, bisogna inserire una istruzione di salto, che permetta alla **CPU** di scavalcare i dati e di raggiungere la prima istruzione da eseguire. Le istruzioni di salto, verranno illustrate in un capitolo successivo.
In questo capitolo, viene utilizzato un altro metodo, molto semplice, che consiste nell'inserire le definizioni dei dati statici, subito dopo le istruzioni di terminazione del programma; quest'area, non potra' mai essere raggiunta (in modo involontario) dalla **CPU**.

Sulla base delle considerazioni appena esposte, siamo finalmente in grado di definire la struttura generale di un programma **Assembly**, destinato ad essere convertito in un eseguibile in formato **COM**; la Figura 17, illustra un modello che verra' utilizzato in tutta la sezione **Assembly Base**, per scrivere esempi di programma **Assembly** in formato **COM**.

Figura 17 - Modello di programma Assembly in formato COM

```
;--------------------------------------------------;
; Nome del modulo COM                              ;
; Descrizione del programma                        ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                          ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

;################ segmento unico #################

COMSEGM     SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM

   org      0100h             ; libera 256 byte per il PSP

start:                        ; entry point

;------- inizio blocco principale istruzioni ------

;-------- fine blocco principale istruzioni -------

   mov      ah, 4ch           ; servizio Terminate Program
   mov      al, 00h           ; exit code = 0
   int      21h               ; chiama i servizi DOS

;----- inizio definizione variabili statiche ------

;------- fine definizione variabili statiche ------

COMSEGM     ENDS

;##################################################

   END      start
```

Il modello presentato in Figura 17, utilizza un segmento unico chiamato **COMSEGM**; per la classe del segmento, e' stata utilizzata la stringa **'CODE'**, ma naturalmente, e' possibile utilizzare anche altre stringhe come **'UNICO'**, **'COMCLASS'**, etc.
Ovviamente, in presenza di un unico segmento, un programma in formato **COM** non puo' superare la dimensione massima prevista per i segmenti di memoria, e cioe' **64** Kb; questa dimensione massima, comprende anche i **256** byte riservati al **PSP**. In realta', sarebbe anche possibile superare questa barriera; in tal caso pero', si andrebbe incontro a complicazioni tali da rendere preferibile la scelta del formato **EXE**.
Anche nel caso dei programmi in formato **COM**, e' possibile spezzare l'unico segmento presente, in due o piu' parti che possono trovarsi distribuite, nello stesso modulo, o anche in moduli differenti; tutte queste parti pero', devono avere lo stesso nome e gli stessi identici attributi, in modo che il

linker ottenga alla fine un unico segmento di programma (ovviamente, l'attributo **Combinazione** non deve essere **PRIVATE**). Se non si seguono queste regole, si ottiene un errore del linker; la suddivisione dei programmi **Assembly** in due o piu' moduli, verra' analizzata in un capitolo successivo.

Tornando alla Figura 17, notiamo la presenza della direttiva:
```
assume cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM
```
Con questa direttiva, stiamo dicendo all'assembler che in presenza di istruzioni del tipo:
```
mov ax, varWord
```
l'identificatore **varWord** rappresenta una componente **Offset** che deve essere associata ad una componente **Seg** contenuta indifferentemente in uno qualunque dei registri di segmento **CS**, **DS**, **ES** e **SS**; infatti, come vedremo tra breve, in presenza di un unico segmento di programma, tutti i registri di segmento verranno inizializzati con la stessa identica componente **Seg**.
Come conseguenza della precedente direttiva **ASSUME**, subito dopo l'**entry point** dovrebbero essere presenti le classiche istruzioni per l'inizializzazione di **DS** e **ES**; se proviamo pero' a scrivere ad esempio:
```
mov ax, COMSEGM
```
otteniamo un errore del linker, rappresentato da un messaggio del tipo:
```
Cannot generate COM file: segment-relocatable items present in module
nomefile.com
```
Con questo messaggio, il linker ci sta dicendo che in un programma in formato **COM**, e' proibita la presenza di istruzioni che fanno riferimento ad una qualsiasi componente **Seg** rilocabile.
Il perche' di questo messaggio e' abbastanza ovvio; in assenza dell'**header**, non esiste nemmeno la **relocation table**. Di conseguenza, il **SO** non e' in grado di modificare il codice macchina di una istruzione che fa' riferimento ad una componente **Seg** rilocabile; e' proibito quindi scrivere anche istruzioni del tipo:
```
mov dx, SEG start
```
E' perfettamente lecito invece scrivere istruzioni del tipo:
```
mov bx, OFFSET start
```
oppure:
```
mov ax, ds
```
Come vedremo tra breve, in un programma in formato **COM**, tutte le inizializzazioni dei registri di segmento **CS**, **DS**, **ES** e **SS**, spettano rigorosamente al **SO**; le considerazioni appena esposte, ci danno un'idea delle notevoli limitazioni presenti in questo tipo di programma eseguibile.

Un altro aspetto abbastanza evidente nel segmento **COMSEGM** di Figura 17, riguarda l'apparente mancanza di un'area riservata allo stack; anche a questo proposito, vedremo che l'inizializzazione standard dello stack di un programma **COM**, spetta al **SO**.

13.11 Esempio di programma Assembly in formato COM

Come esempio pratico, trasformiamo in formato **COM** il programma di Figura 1; il risultato di questa trasformazione, ci porta ad ottenere il file **COMTEST.ASM** illustrato in Figura 18.

Figura 18 - Esempio di programma Assembly in formato COM

```
;--------------------------------------------------;
; File comtest.asm                                 ;
; Esempio di programma Assembly in formato COM     ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

; dichiarazione record RecData

RecData RECORD Giorno: 5, Mese: 4, Anno: 11

; dichiarazione struttura Rect

Point2d STRUC

   x     dw    ?
   y     dw    ?

Point2d ENDS

Rect STRUC

   p1    Point2d  < ?, ? >
   p2    Point2d  < ?, ? >

Rect ENDS

; dichiarazione struttura Automobile

Automobile  STRUC

   CodModello  dw       ?
   Cilindrata  dw       ?
   Cavalli     dw       ?
      ALIGN    4
   Revisioni   RecData  10 DUP ( < > )

Automobile  ENDS

;################ segmento unico #################

COMSEGM     SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM

   org      0100h                   ; libera 256 byte per il PSP

start:                              ; entry point

;------- inizio blocco principale istruzioni ------
```

```
; inizializzazione di varRect

   mov      varRect.p1.x, 3500
   mov      varRect.p1.y, 8000
   mov      varRect.p2.x, 6850
   mov      varRect.p2.y, 9700

; inizializzazione di vettRect[0] (elemento di indice 0)

   mov      vettRect[0].p1.x, 3cf2h
   mov      vettRect[0].p1.y, 8dabh
   mov      vettRect[0].p2.x, 5c2bh
   mov      vettRect[0].p2.y, 66f1h

; inizializzazione di vettRect[8] (elemento di indice 1)

   mov      vettRect[8].p1.x, 93f1h
   mov      vettRect[8].p1.y, 359ch
   mov      vettRect[8].p2.x, 86ffh
   mov      vettRect[8].p2.y, 12bch

; inizializzazione di FiatPanda

   mov      FiatPanda.CodModello, 3518
   mov      FiatPanda.Cilindrata, 1000
   mov      FiatPanda.Cavalli, 45

; inizializzazione di FiatPanda.Revisioni[0] con la data
; 12/3/2008 = 1100b/00011b/11111011000b

   mov      FiatPanda.Revisioni[0], 1100000111111011000b

; inizializzazione di FiatPanda.Revisioni[4] con la data
; 12/3/2010 = 1100b/00011b/11111011010b

   mov      FiatPanda.Revisioni[4], 1100000111111011010b

; inizializzazione di vettAuto[0] (elemento di indice 0)

   mov      vettAuto[0].CodModello, 2243
   mov      vettAuto[0].Cilindrata, 2000
   mov      vettAuto[0].Cavalli, 120
   mov      vettAuto[0].Revisioni[0], 1100000111111011000b
   mov      vettAuto[0].Revisioni[4], 1100000111111011010b
   mov      vettAuto[0].Revisioni[8], 1100000111111011100b

; inizializzazione di vettAuto[46] (elemento di indice 1)

   mov      vettAuto[46].CodModello, 2244
   mov      vettAuto[46].Cilindrata, 2500
   mov      vettAuto[46].Cavalli, 170
   mov      vettAuto[46].Revisioni[0], 1100000111111011000b
   mov      vettAuto[46].Revisioni[4], 1100000111111011010b
   mov      vettAuto[46].Revisioni[8], 1100000111111011100b

; somma a 32 bit tra varDword e DataViaggio

   mov      eax, varDword
   add      eax, DataViaggio

; somma a 16 bit tra varWord e i primi 16 bit di DataViaggio
```

```
   mov      ax, varWord
   add      ax, word ptr DataViaggio[0]

; somma a 16 bit tra varWord e i secondi 16 bit di DataViaggio

   mov      ax, varWord
   add      ax, word ptr DataViaggio[2]

; somma a 8 bit tra varByte e i primi 8 bit di DataViaggio

   mov      al, varByte
   add      al, byte ptr DataViaggio[0]

; somma a 8 bit tra varByte e i secondi 8 bit di DataViaggio

   mov      al, varByte
   add      al, byte ptr DataViaggio[1]

; somma a 8 bit tra varByte e i terzi 8 bit di DataViaggio

   mov      al, varByte
   add      al, byte ptr DataViaggio[2]

; somma a 8 bit tra varByte e i quarti 8 bit di DataViaggio

   mov      al, varByte
   add      al, byte ptr DataViaggio[3]

; somma a 32 bit tra FiatPanda.Revisioni[4] e vettAuto[46].Revisioni[8]

   mov      eax, FiatPanda.Revisioni[4]
   add      eax, vettAuto[46].Revisioni[8]

; somma a 32 bit tra varDword e vettAuto[46].Revisioni[8]

   mov      eax, varDword
   add      eax, vettAuto[46].Revisioni[8]

; somma a 16 bit tra varWord e i primi 16 bit di vettAuto[46].Revisioni[8]

   mov      ax, varWord
   add      ax, word ptr vettAuto[46].Revisioni[8+0]


; somma a 16 bit tra varWord e i secondi 16 bit di vettAuto[46].Revisioni[8]

   mov      ax, varWord
   add      ax, word ptr vettAuto[46].Revisioni[8+2]

; somma a 8 bit tra varByte e i primi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+0]

; somma a 8 bit tra varByte e i secondi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+1]

; somma a 8 bit tra varByte e i terzi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+2]
```

```
; somma a 8 bit tra varByte e i quarti 8 bit di vettAuto[46].Revisioni[8]

   mov      al, varByte
   add      al, byte ptr vettAuto[46].Revisioni[8+3]

; somma a 8 bit tra i secondi 8 bit di FiatPanda.Revisioni[4] e
; i terzi 8 bit di vettAuto[46].Revisioni[8]

   mov      al, byte ptr FiatPanda.Revisioni[4+1]
   add      al, byte ptr vettAuto[46].Revisioni[8+2]

;-------- fine blocco principale istruzioni -------

   mov      ah, 4ch                ; servizio Terminate Program
   mov      al, 00h                ; exit code = 0
   int      21h                    ; chiama i servizi DOS

;----- inizio definizione variabili statiche ------

   ALIGN    4                                   ; allinea alla DWORD
varByte     db          0bh                     ; intero a 1 byte
   ALIGN    4                                   ; allinea alla DWORD
varWord     dw          0abcdh                  ; intero a 2 byte
   ALIGN    4                                   ; allinea alla DWORD
varDword    dd          12345678h               ; intero a 4 byte
varQword    dq          123456789abcdef0h       ; intero a 6 byte
varTbyte    dt          00001111222233334444h   ; intero a 10 byte
   ALIGN    4                                   ; allinea alla DWORD
varFloat32  dd          3.14                    ; reale a 4 byte
varFloat64  dq          -5.12345E-200           ; reale a 8 byte

DataViaggio RecData     < 12, 06, 2001 >        ; RecData a 4 byte
varRect     Rect        < >                     ; Rect a 8 byte
vettRect    Rect        10 dup ( < > )          ; vettore di 10 Rect
FiatPanda   Automobile  < >                     ; Automobile a 46 byte
vettAuto    Automobile  20 dup ( < > )          ; vettore di 20 Automobile

;------- fine definizione variabili statiche ------

COMSEGM     ENDS

;###################################################

   END      start
```

Notiamo subito la presenza di alcune differenze importanti rispetto al codice sorgente di Figura 1; in Figura 18, e' stata utilizzata la direttiva **ALIGN**, sia per i membri della struttura **Automobile**, sia per l'allineamento alla **DWORD** dei dati statici del programma. Notiamo anche la presenza di una direttiva **ALIGN 4**, che precede l'inizio delle definizioni dei dati statici; questa direttiva e' necessaria in quanto, molto probabilmente, l'ultima istruzione del blocco codice principale termina ad un offset che provoca il disallineamento dell'informazione successiva. Per i programmi in formato **COM**, si consiglia di ricorrere sempre a questo accorgimento.

13.11.1 Assembling & Linking di un programma in formato COM

L'assemblaggio di **COMTEST.ASM** avviene nel solito modo; nel caso del **TASM**, il comando da impartire e':

```
..\bin\tasm /l comtest.asm
```

Nel caso del **MASM**, il comando da impartire e':

```
..\bin\ml /c /Fl comtest.asm
```

I comandi appena impartiti, generano un **object file** chiamato **COMTEST.OBJ**, e anche un **listing file** chiamato **COMTEST.LST**; grazie al **listing file**, possiamo constatare che il blocco **COMSEGM**, occupa complessivamente **06BCh** byte (**1724** byte), e comprende anche i **256** byte del **PSP**.
Sempre attraverso il **listing file**, si puo' notare che l'**entry point**, rappresentato dall'etichetta **start**, si trova all'offset **0100h**; inoltre, tutti i dati definiti in **COMSEGM**, si trovano allineati alla **DWORD** (grazie anche al fatto che questa volta, ogni struttura **Automobile** occupa **48** byte).

Passiamo ora alla fase di linking; il comando da impartire con il **TASM** e':

```
..\bin\tlink /t comtest.obj
```

Il comando da impartire con il **MASM** e':

```
..\bin\link /map /tiny comtest.obj
```

In assenza dell'opzione **/t** per il **TASM** e **/tiny** per il **MASM**, il linker genera un normale eseguibile in formato **EXE**, chiamato **COMTEST.EXE**; in questo caso, viene generato anche un messaggio di avvertimento (**warning**) che ci informa dell'assenza dello stack.
In presenza invece dell'opzione **/t** per il **TASM** e **/tiny** per il **MASM**, il linker genera un file eseguibile chiamato **COMTEST.COM**; in questo caso, non viene generato nessun warning relativo all'assenza dello stack. Oltre all'eseguibile in formato **COM**, viene generato anche un **map file** chiamato **COMTEST.MAP**, che ci fornisce un riassunto semplificato del lavoro svolto dal linker.

13.11.2 Caricamento in memoria ed esecuzione di un programma in formato COM

Vediamo ora quello che succede, quando dal prompt del **DOS**, si impartisce il comando:

```
comtest
```

Dopo aver seguito tutta la procedura gia' descritta a proposito dell'esempio **EXETEST.EXE**, il **loader** del **SO** trova un file chiamato **COMTEST.COM**; non trovando nessuna stringa **'MZ'** all'inizio di questo file, il **loader** avvia le procedure standard per il caricamento in memoria di un eseguibile in formato **COM**.

Come al solito, il **DOS** alloca due blocchi di memoria, uno per l'**environment segment**, e l'altro per il **program segment**; la domanda che ci poniamo e': in base a quali criteri il **DOS** determina la dimensione del **program segment**?
La tecnica utilizzata dal **DOS** e' molto semplice; al **program segment** di un qualunque programma in formato **COM**, viene riservato un intero segmento di memoria che, come sappiamo, occupa **64** Kb ed e' sempre allineato al paragrafo.
Supponiamo allora che il **DOS** abbia riservato a **COMTEST.COM**, il segmento di memoria che parte dall'indirizzo fisico **01BC0h**; a questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **01BCh:0000h**.
A questo punto, il **DOS** legge dal disco il file **COMTEST.COM** (che contiene il solo modulo caricabile), e lo copia tale e quale nel **program segment**; nei primi **256** byte (**0100h**) che precedono l'**entry point**, viene sistemato il **PSP**. Possiamo dire quindi che il **PSP**, occupa tutti gli indirizzi fisici del **program segment**, compresi tra **01BC0h** e:

```
01BC0h + (0100h - 1h) = 01CBFh
```

In termini di indirizzi logici, il **PSP** viene inserito nel segmento di memoria n. **01BCh**, e occupa tutti gli offset di questo segmento, compresi tra **0000h** e:

```
0000h + (0100h - 1h) = 00FFh
```

In sostanza, il **PSP** e' compreso tra gli indirizzi logici **01BCh:0000h** e **01BCh:00FFh**.

L'inizio del **PSP** coincide con l'inizio del **program segment**; anche nei programmi **COM** quindi, per convenzione, si utilizza la sigla **PSP**, per indicare la componente **Seg** dell'indirizzo logico iniziale (**01BCh:0000h**) dello stesso **program segment**. Possiamo scrivere quindi:

```
PSP = 01BCh
```

Il **DOS** passa ora alla inizializzazione standard dei registri della **CPU**; per i registri di segmento **CS**, **DS**, **ES** e **SS**, il **DOS** pone ovviamente:

```
CS = DS = ES = SS = PSP = 01BCh
```

In sostanza, il blocco codice, il blocco dati e il blocco stack, partono tutti dallo stesso indirizzo fisico **01BC0h**.
Il **PSP** parte dall'indirizzo logico **01BCh:0000h**, per cui l'**entry point** del programma, si viene a trovare all'indirizzo logico **01BCh:0100h**; il **DOS** pone quindi:

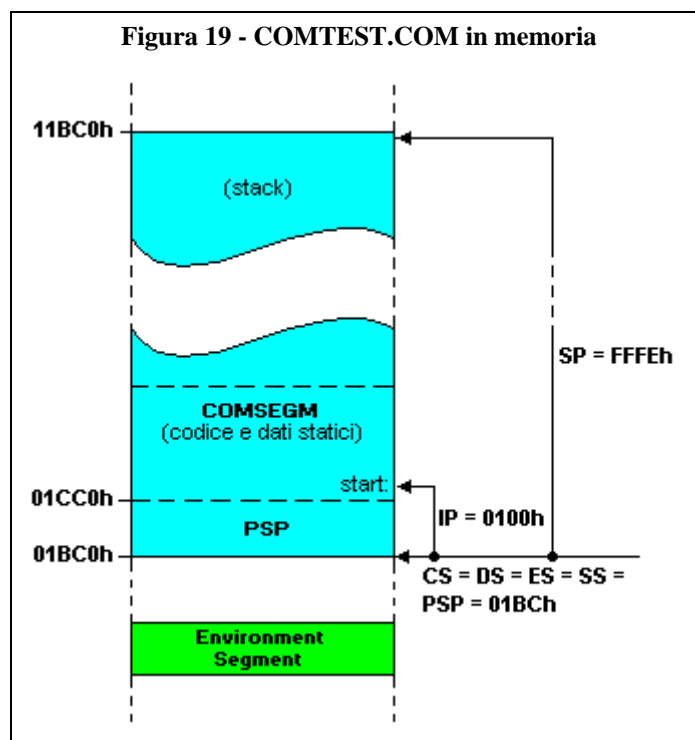
```
CS:IP = 01BCh:0100h
```

Il registro **SP** viene inizializzato con l'offset **FFFEh** (che e' il piu' grande offset pari, all'interno del segmento di memoria n. **01BCh**); per la coppia **SS:SP**, il **DOS** pone quindi:

```
SS:SP = 01BCh:FFFEh
```

Successivamente, il **DOS** inserisce una **WORD** di valore **0000h**, all'indirizzo logico **SS:SP** (cioe' **01BCh:FFFEh**); il perche' di questa **WORD** verra' chiarito in un prossimo capitolo.

A questo punto, il programma e' pronto per l'esecuzione; in seguito a tutto questo lavoro svolto dal **DOS**, il programma **COMTEST.COM** assume in memoria la disposizione illustrata in Figura 19.

Figura 19 - COMTEST.COM in memoria

Come possiamo notare, il programma **COMTEST.COM** ha a sua disposizione una notevole quantita' di memoria per lo stack; in base al fatto che il **PSP**, il codice e i dati statici di questo programma, occupano **06BCh** byte, possiamo dire che lo spazio riservato allo stack e' pari a:

```
FFFEh - 06BCh = 65534 - 1724 = 63810 byte
```

E' necessario ricordare che, man mano che lo stack si riempie, il contenuto del registro **SP** viene via via decrementato; se lo spazio riservato allo stack e' troppo piccolo, c'e' il rischio che **SP** invada l'area riservata al codice e ai dati statici, che verrebbero cosi' sovrascritti dai dati temporanei. La Figura 19, permette anche di capire come ci si deve regolare, nel momento in cui si vuole creare un segmento misto **dati + stack**; in sostanza, e' fondamentale che l'area riservata allo stack, venga disposta sempre nella parte finale del segmento misto.

Il **PSP**, il codice, i dati statici e lo stack di **COMTEST.COM**, occupano l'intero segmento di memoria n. **01BCh**; un segmento di memoria e' formato da **65536** byte (**10000h** byte). Possiamo dire quindi che l'indirizzo fisico iniziale del segmento di memoria successivo al nostro programma, sara':

```
01BC0h + 10000h = 11BC0h
```

Una volta che il **DOS** ha inizializzato i necessari registri della **CPU**, puo' partire la fase di esecuzione di **COMTEST.COM**; il controllo passa quindi alla stessa **CPU** che trova la coppia **CS:IP** inizializzata con l'**entry point 01BCh:0100h**. Da questo punto in poi, la fase di esecuzione si svolge nello stesso modo gia' descritto per **EXETEST.EXE**.

Nel prossimo capitolo, verranno illustrati diversi strumenti, che ci permetteranno di verificare in pratica, tutto cio' che e' stato appena esposto in relazione alla struttura interna di un file eseguibile, e al comportamento in memoria di un programma in esecuzione.

Capitolo 14 - Disassembling

Nel precedente capitolo, sono stati esposti numerosi concetti teorici, relativi al lavoro svolto dall'assembler, dal linker e dal **SO**, nel processo di conversione del codice sorgente **Assembly** in codice eseguibile; in questo capitolo invece, questi stessi concetti teorici verranno verificati in pratica attraverso l'analisi di un programma eseguibile.
Questo tipo di analisi, puo' essere effettuata in diversi modi e con diversi strumenti, dai piu' semplici ai piu' sofisticati; nel nostro caso, analizzeremo in modo sintetico, alcune tecniche che ci permetteranno di conoscere numerosi dettagli interni di un programma eseguibile in formato **EXE** o **COM**.

14.1 Analisi di un file eseguibile con un editor binario

Un qualunque file, viene memorizzato su disco sotto forma di sequenza di byte, cioe' sotto forma di valori binari da **8** bit ciascuno; proprio per questo motivo, i file memorizzati su disco vengono definiti genericamente, **file binari**.
I linguaggi di programmazione di alto livello, tendono invece a distinguere tra **file binari** e **file di testo**; come si puo' facilmente intuire, questa distinzione e' puramente formale. Consideriamo, ad esempio, la seguente sequenza di byte memorizzata in un file di testo:
`41h, 73h, 73h, 65h, 6Dh, 62h, 6Ch, 79h, 0Dh, 0Ah`
Un editor di testo, nella fase di apertura di questo file, cerchera' di convertire ogni byte, nel simbolo associato al corrispondente codice ASCII; i simboli cosi' ottenuti, verranno poi visualizzati sullo schermo. Come si puo' facilmente constatare, i primi **8** byte codificano la stringa:
`Assembly`
Il penultimo byte (**0Dh**), rappresenta un codice che permette di simulare sullo schermo del computer, il ritorno carrello (**carriage return**) della macchina da scrivere; un editor di testo, interpreta questo codice spostando il cursore lampeggiante, sul bordo sinistro dello schermo.
L'ultimo byte (**0Ah**), rappresenta un codice che permette di simulare sullo schermo del computer, l'avanzamento riga (**line feed**) della macchina da scrivere; un editor di testo, interpreta questo codice spostando il cursore lampeggiante, sulla riga successiva dello schermo.

Appare evidente il fatto che questo file di testo, quando viene memorizzato su disco, risulta disposto sotto forma di sequenza di byte, come un qualsiasi file binario; in sostanza, i file di testo sono particolari file binari, che contengono una sequenza di codici ASCII.

Se proviamo ad aprire un file eseguibile con un editor di testo, otteniamo sullo schermo una sequenza incomprensibile di simboli; l'editor di testo infatti, cerchera' di trattare come codici ASCII, quelli che, in realta', sono i codici macchina dei dati e delle istruzioni di un programma.
Se vogliamo analizzare il contenuto grezzo di un file binario, possiamo servirci di appositi editor chiamati **editor binari**; l'editor binario, visualizza i vari byte che formano un file binario, senza attribuire a questi byte nessun significato.
Un esempio pratico di editor binario, e' rappresentato dal programma **QEDITOR.EXE** che viene fornito con **MASM32**; questo editor, oltre a visualizzare i file di testo, dispone anche del menu:
`File - Open Binary as Hex`

Proviamo allora ad aprire con questo menu, il file **EXETEST.EXE** che abbiamo ottenuto nel precedente capitolo; la Figura 1 mostra la parte iniziale delle informazioni visualizzate da **QEDITOR**.

Figura 1 - Dump di EXETEST.EXE

```
; c:\tasm\asmbase\exetest.exe  1935 bytes

00000000: 4D 5A 8F 01 04 00 01 00 - 20 00 41 00 FF FF 59 00
00000010: 00 04 00 00 08 00 44 00 - 3E 00 00 00 01 00 FB 71
00000020: 6A 72 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000030: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 09 00
00000040: 44 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000050: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000060: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000070: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
```

Questo particolare tipo di visualizzazione delle informazioni binarie, presenti in un qualsiasi supporto di memoria (disco, **RAM**, etc), viene anche definito **memory dump**.
Osserviamo subito che **QEDITOR** visualizza il **dump** sotto forma di righe, dove ogni riga contiene un gruppo di **16** byte (cioe' **1** paragrafo); i vari byte vengono disposti da sinistra verso destra, per cui i dati di tipo **WORD**, **DWORD**, etc, appaiono disposti al contrario rispetto alla loro rappresentazione in notazione posizionale. Notiamo anche che **QEDITOR**, mostra nella parte sinistra, gli offset delle informazioni presenti nel file binario che abbiamo aperto; questi offset sono degli spiazzamenti calcolati, ovviamente, rispetto all'inizio del file.

Ricordando quanto e' stato detto nel precedente capitolo, possiamo subito intuire che le informazioni presenti in Figura 1, si riferiscono all'**header** di **EXETEST.EXE**; infatti, si nota subito che i primi due byte (**4Dh** e **5Ah**), sono i codici ASCII della stringa **'MZ'**. Proviamo allora ad "estrarre" dalla Figura 1, la struttura di controllo dell'**header** di **EXETEST.EXE**; il risultato che si ottiene, viene mostrato in Figura 2.

Figura 2 - Struttura di controllo dell'header di EXETEST.EXE

```
Signature                  = 'MZ'
LastPageSize               = 018Fh
PageCount                  = 0004h
RelocationRecords          = 0001h
HeaderParagraphs           = 0020h
MinimumAllocation          = 0041h
MaximumAllocation          = FFFFh
InitialSS                  = 0059h
InitialSP                  = 0400h
CheckSum                   = 0000h
InitialIP                  = 0008h
InitialCS                  = 0044h
StartOfRelocationTable     = 003Eh
OverlayNumber              = 0000h
```

Le informazioni di Figura 2, confermano l'esattezza di tutti i calcoli che avevamo raccolto nella Figura 9 del Capitolo 13; dalla Figura 2 ricaviamo anche:
`RelocationRecords = 0001h`
e:
`StartOfRelocationTable = 003Eh`
Spostandoci infatti all'offset **003Eh** del blocco di Figura 1, troviamo le seguenti informazioni:
`RelocationOffset = 0009h` e `RelocationSeg = 0044h`
Anche in questo caso, vengono confermati i calcoli riportati nella Figura 9 del Capitolo 13.

Proviamo ora a calcolare le dimensioni di **EXETEST.EXE**; in base ai dati di Figura 2, si ottiene:

```
((PageCount - 1) * 512) + LastPageSize = (3 * 512) + 018Fh = 1536 + 399 = 1935 
byte
```

Infatti, nella cartella **ASMBASE** possiamo constatare che il file **EXETEST.EXE** occupa proprio **1935** byte.

Sempre dalla Figura 2 ricaviamo:

```
HeaderParagraphs = 0020h
```

Cio' significa che l'**header**, assume la dimensione minima possibile, pari a:

```
0020h * 10h = 0200h = 512 byte
```

Di conseguenza, all'offset **0200h** di **EXETEST.EXE**, dovremmo trovare l'inizio di **DATASEGM**; a tale proposito, osserviamo il contenuto della Figura 3.

Figura 3 - Dump di EXETEST.EXE

```
000001F0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000200: 0B CD AB 78 56 34 12 F0 - DE BC 9A 78 56 34 12 44
00000210: 44 33 33 22 22 11 11 00 - 00 C3 F5 48 40 A8 F0 E7
00000220: 20 BD 5F 8F 96 D1 37 06 - 00 00 00 00 00 00 00 00
00000230: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000240: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000250: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
00000260: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
```

Ci accorgiamo subito che le informazioni presenti in Figura 3, a partire dall'offset **0200h**, rappresentano proprio i dati visibili nel **listing File** della Figura 2 del Capitolo 13 (righe dalla **52** alla **65**).
Sempre dal **listing file**, rileviamo che **DATASEGM** occupa **0447h** byte; inoltre, sappiamo che subito dopo la fine di **DATASEGM**, il linker ha inserito un buco di memoria da **1** byte per poter allineare **CODESEGM** alla **DWORD**. Di conseguenza, **CODESEGM** dovrebbe iniziare dall'offset:

```
0200h + 0447h + 1h = 0648h
```

del file **EXETEST.EXE**; a tale proposito, osserviamo il contenuto della Figura 4.

Figura 4 - Dump di EXETEST.EXE

```
00000640: 00 00 00 00 00 00 00 00 - B8 00 00 8E D8 C7 06 29
00000650: 00 AC 0D C7 06 2B 00 40 - 1F C7 06 2D 00 C2 1A C7
00000660: 06 2F 00 E4 25 C7 06 31 - 00 F2 3C C7 06 33 00 AB
00000670: 8D C7 06 35 00 2B 5C C7 - 06 37 00 F1 66 C7 06 39
00000680: 00 F1 93 C7 06 3B 00 9C - 35 C7 06 3D 00 FF 86 C7
00000690: 06 3F 00 BC 12 C7 06 81 - 00 BE 0D C7 06 83 00 E8
000006A0: 03 C7 06 85 00 2D 00 66 - C7 06 87 00 D8 1F 0C 00
000006B0: 66 C7 06 8B 00 DA 1F 0C - 00 C7 06 AF 00 C3 08 C7
```

Anche in questo caso, ci accorgiamo subito che le informazioni presenti in Figura 4, a partire dall'offset **0648h**, rappresentano proprio i codici macchina delle istruzioni presenti nel blocco **CODESEGM**; notiamo inoltre che il buco di memoria inserito dal linker all'offset **0647h**, e' un byte di valore **00h**.

Se ora ci portiamo alla fine del file **EXETEST.EXE**, troviamo gli ultimi due byte, che valgono **CDh**, **21h**; ma questo, non e' altro che il codice macchina dell'istruzione:

```
INT 21h
```

che termina il nostro programma!
Tutto cio' significa che nel modulo caricabile presente nel file **EXETEST.EXE**, il linker non ha inserito i **0400h** byte del blocco **STACKSEGM**; come si spiega questa situazione?

Per rispondere a questa domanda, bisogna ricordare che il blocco **STACKSEGM** si trova proprio alla fine del nostro programma, e contiene **0400h** byte di dati non inizializzati; in un caso di questo genere, il linker elimina questo segmento di programma dal modulo caricabile, e inizializza opportunamente il campo **MinimumAllocation** presente nella struttura di controllo dell'**header**.
Osservando infatti la Figura 2, ci accorgiamo che:

```
MinimumAllocation = 0041h
```

Questo valore e' espresso in paragrafi, e rappresenta quindi:

```
0041h * 10h = 0410h = 1040 byte
```

di memoria aggiuntiva; tale memoria, viene appunto aggiunta dal **DOS**, al **program segment** destinato a **EXETEST.EXE**.
Si puo' notare che il valore **1040**, e' leggermente superiore a quello che noi avevamo richiesto (**1024** byte per lo stack); questo accade in quanto il linker, tiene conto del fatto che anche dopo la fine di **CODESEGM**, e' previsto un buco di memoria da **1** byte, che garantisce l'allineamento di **STACKSEGM** al paragrafo. Bisogna ricordare, inoltre, che i blocchi di memoria **DOS** hanno sempre una dimensione in byte multipla intera di **16**; nel nostro caso, per poter contenere **1** byte di allineamento, piu' **1024** byte di stack, abbiamo bisogno, come minimo, di un blocco di memoria aggiuntiva formato appunto da **41h** paragrafi (**65** paragrafi), pari a:

```
65 * 16 = 1040 byte
```

Come si intuisce dal nome, la memoria aggiuntiva puo' essere posizionata solo dopo la fine del **program segment**; di conseguenza, se disponiamo **STACKSEGM** prima di **CODESEGM** o di **DATASEGM**, il linker provvede ad incorporare anche lo stack, direttamente all'interno del modulo caricabile.

Attraverso gli editor binari, e' possibile analizzare anche alcune stranezze relative alle dimensioni dei file eseguibili in formato **COM**; consideriamo a tale proposito il file **COMTEST.COM** che abbiamo creato nel precedente capitolo.
In base al **listing file** prodotto dall'assembler, si rileva che il blocco **COMSEGM** occupa **06BCh** byte (cioe' **1724** byte) che comprendono anche i **256** byte del **PSP**; nella cartella **ASMBASE** notiamo pero' che il file **COMTEST.COM** (che contiene solo il modulo caricabile), occupa **1468** byte!
Se proviamo ad aprire il file **COMTEST.COM** con un editor binario, ci accorgiamo che mancano i **256** byte del **PSP**; come si nota infatti in Figura 5, il file inizia direttamente con i codici macchina delle varie istruzioni.

Figura 5 - Dump di COMTEST.COM

```
; c:\tasm\asmbase\comtest.com  1468 bytes

00000000: C7 06 74 02 AC 0D C7 06 - 76 02 40 1F C7 06 78 02
00000010: C2 1A C7 06 7A 02 E4 25 - C7 06 7C 02 F2 3C C7 06
00000020: 7E 02 AB 8D C7 06 80 02 - 2B 5C C7 06 82 02 F1 66
00000030: C7 06 84 02 F1 93 C7 06 - 86 02 9C 35 C7 06 88 02
00000040: FF 86 C7 06 8A 02 BC 12 - C7 06 CC 02 BE 0D C7 06
00000050: CE 02 E8 03 C7 06 D0 02 - 2D 00 66 C7 06 D4 02 D8
00000060: 1F 0C 00 66 C7 06 D8 02 - DA 1F 0C 00 C7 06 FC 02
00000070: C3 08 C7 06 FE 02 D0 07 - C7 06 00 03 78 00 66 C7
```

Come ulteriore verifica, notiamo che:

```
1724 - 1468 = 256 byte
```

Tutto cio' e' una diretta conseguenza del procedimento standard utilizzato dal linker e dal **SO**, per la gestione dei programmi in formato **COM**; il linker ha ricevuto (da noi) l'ordine di generare un file **COM**, e quindi inserisce nel file stesso, solo il codice e i dati statici. Il linker si comporta in questo modo, perche' sa' che il **SO**, non trovando la stringa **'MZ'** all'inizio del file **COMTEST.COM**, lo trattera' come un eseguibile in formato **COM**, e avviera' la procedura standard per il suo

caricamento in memoria; questa procedura prevede anche la creazione, all'inizio del **program segment**, dello spazio necessario per il **PSP**, con conseguente slittamento dell'**entry point**, all'offset **0100h**.

Con l'ausilio di un editor binario, si possono fare parecchie cose interessanti; ad esempio, un programmatore **Assembly** piuttosto esperto, potrebbe apportare delle modifiche ad un file eseguibile. Bisogna prestare molta attenzione al fatto che con **QEDITOR**, se si vuole salvare su disco il contenuto di un file binario, ci si deve servire del menu:
`File - Save Hex As Binary`
Questo menu, salva su disco, solamente le informazioni binarie del file; tutte le altre informazioni (commenti e offset), vengono eliminate. Se proviamo a servirci del normale menu:
`File - Save`
otteniamo un file corrotto che, in fase di esecuzione, provoca un sicuro crash; cio' accade in quanto, con il normale menu **Save**, **QEDITOR** salva su disco, non solo le informazioni binarie del file, ma anche le altre informazioni aggiuntive, come ad esempio, gli offset e i commenti visibili in Figura 5.

Facciamo un semplice esperimento, che consiste nel creare direttamente con **QEDITOR**, un file binario chiamato, ad esempio, **TEST.COM**; il programma che si ottiene, visualizza una stringa attraverso il servizio n. **09h** (**Display String**) della **INT 21h**. Questo servizio presenta le caratteristiche illustrate in Figura 6.

Figura 6 - Servizio n. 09h della INT 21h

```
INT 21h - Servizio n. 09h - Display String:
   visualizza una stringa sullo schermo, a partire dalla posizione
   corrente del cursore

Argomenti richiesti:
   AH = 09h
   DS:DX = indirizzo logico Seg:Offset della stringa

Note:
   la stringa deve terminare con il simbolo '$' (codice ASCII 24h)
```

Il programma, in formato **COM**, contiene le istruzioni mostrate in Figura 7.

Figura 7 - Programma TEST.COM

```
COMSEGM  SEGMENT PARA PUBLIC USE16 'CODE'

   org      0100h
start:
   mov      dx, offset messaggio    ; 0100h  BAh 010Dh
   mov      ah, 09h                 ; 0103h  B4h 09h
   int      21h                     ; 0105h  CDh 21h
   mov      ah, 4ch                 ; 0107h  B4h 4Ch
   mov      al, 00h                 ; 0109h  B0h 00h
   int      21h                     ; 010Bh  CDh 21h
messaggio   db    'Prova', '$'      ; 010Dh  50h 72h 6Fh 76h 61h 24h

COMSEGM  ENDS

   END      start
```

Come si nota in Figura 6, **DS:DX** deve contenere l'indirizzo logico **Seg:Offset** della stringa da visualizzare; la componente **Seg** di questo indirizzo e' **COMSEGM**, per cui **DS** non necessita di nessuna modifica (infatti, nel formato **COM** il **DOS** pone **DS=COMSEGM**). Osservando la

struttura del nostro programma **COM**, abbiamo anche la certezza che l'offset **010Dh** di **messaggio**, non subira' nessuna rilocazione; di conseguenza, possiamo caricare in **DX** direttamente il valore **010Dh**.
Apriamo ora **QEDITOR.EXE**, e inseriamo direttamente il codice macchina del nostro programma (bisogna rispettare rigorosamente, le regole di formattazione seguite da **QEDITOR**); si ottiene cosi' la seguente situazione:

```
00000000: BA 0D 01 B4 09 CD 21 B4 - 4C B0 00 CD 21 50 72 6F
00000010: 76 61 24
```

Attraverso il menu **Save Hex as Binary** di **QEDITOR**, salviamo il nostro file da **19** byte, chiamandolo **TEST.COM**; possiamo subito notare, che il file **TEST.COM** salvato su disco, e' formato proprio da **19** byte. Eseguendo ora **TEST.COM**, possiamo constatare che il programma che abbiamo appena creato "a mano", e' perfettamente funzionante; sullo schermo, verra' infatti mostrata la stringa:

```
Prova
```

Gli editor binari vengono largamente utilizzati anche per analizzare la struttura interna di particolari formati di file; in questo modo, si possono scoprire parecchi segreti relativi, ad esempio, alla codifica dei file audio (**WAV**, **MID**, **MP3**, etc), dei file video (**AVI**, **MPG**, etc), dei file immagine (**GIF**, **BMP**, **JPG**, **TIFF**, etc), e di numerosissimi altri tipi di file.

Se vogliamo effettuare una analisi molto piu' sofisticata di un file eseguibile, possiamo servirci di appositi strumenti, capaci di mostrare sullo schermo, direttamente il codice **Assembly** dell'eseguibile stesso; questa categoria di strumenti, comprende anche i cosiddetti **debugger**.

14.2 Analisi di un file eseguibile con un debugger

Come e' stato spiegato in un precedente capitolo, la corrispondenza biunivoca che esiste tra codice macchina e codice **Assembly**, rende possibile l'esistenza di particolari strumenti chiamati **disassembler** (disassemblatori); il disassembler e' un programma che legge il codice macchina presente in un file eseguibile, e lo converte in codice **Assembly**.
Come si puo' facilmente intuire, uno strumento cosi' potente puo' offrire un aiuto enorme ai programmatori che hanno la necessita' di analizzare la struttura interna e il comportamento di un eseguibile; proprio grazie alle loro notevoli potenzialita', i disassembler vengono largamente usati anche dai pirati informatici, per finalita' poco lecite.

Come e' stato detto in precedenza, nella categoria dei disassembler rientrano anche i **debugger**; lo scopo fondamentale dei debugger, e' quello di aiutare i programmatori, nella complessa fase di ricerca delle cause che provocano il malfunzionamento dei propri programmi.
Un programmatore che vuole sfruttare al massimo le potenzialita' dei debugger, deve creare dei programmi eseguibili, che contengono al loro interno una serie di informazioni, chiamate appunto **informazioni di debugging**; grazie a queste informazioni, il debugger e' in grado di fornire una enorme quantita' di dettagli, relativi a tutto cio' che accade all'interno di un programma nella fase di esecuzione.

Gli utilizzatori degli strumenti di sviluppo della **Microsoft**, devono acquistare separatamente un apposito debugger, chiamato **CodeView**; gli utilizzatori degli strumenti di sviluppo della **Borland** invece, hanno a disposizione un potentissimo debugger chiamato **Turbo Debugger**, rappresentato dal file **TD.EXE** (o **TD32.EXE** per i programmi a **32** bit per **Windows**).
Analizziamo proprio il caso del **Turbo Debugger**; attraverso **TD**, e' possibile "debuggare" qualsiasi programma scritto con gli strumenti di sviluppo della **Borland** (come, **TASM**, **Turbo Pascal**, **Delphi**, **C/C++ Compiler**, etc).

In riferimento, ad esempio, al file **EXETEST.ASM** presentato nel precedente capitolo, dobbiamo effettuare l'assemblaggio attraverso il seguente comando:
```
..\bin\tasm /zi exetest.asm
```
L'opzione **/zi** indica al **TASM** di inserire tutte le necessarie informazioni di debugging all'interno dell'object file **EXETEST.OBJ**.
La fase di linking deve essere effettuata attraverso il comando:
```
..\bin\tlink /v exetest.obj
```
L'opzione **/v** indica a **TLINK** di inserire tutte le necessarie informazioni di debugging all'interno del file eseguibile **EXETEST.EXE**.
A questo punto, possiamo lanciare il debugger con il comando:
```
..\bin\td exetest.exe
```
Una volta entrati nel debugger, abbiamo a disposizione una enorme quantita' di strumenti, attraverso i quali possiamo esaminare nel minimo dettaglio, tutta la fase di esecuzione di **EXETEST.EXE**; chi volesse approfondire questo argomento, puo' consultare il manuale utente fornito insieme al debugger.

Le informazioni di debugging, influiscono negativamente sulla velocita' di esecuzione di un programma, e aumentano notevolmente le dimensioni del codice; proprio per questo motivo, una volta che la fase di debugging e' terminata, e' necessario ricreare l'eseguibile, disabilitando la generazione di tutte le informazioni destinate al debugger.
Sempre nel caso del **TASM**, se vogliamo eliminare da **EXETEST.EXE** qualsiasi informazione di debugging (anche quelle predefinite), dobbiamo effettuare l'assemblaggio attraverso l'opzione **/zn**; il comando da impartire e' quindi:
```
..\bin\tasm /zn exetest.asm
```
A questo punto, possiamo effettuare il linking con il solito comando:
```
..\bin\tlink exetest.obj
```

L'uso dei debugger per la ricerca degli errori nei programmi, non trova molto seguito tra i programmatori **Assembly**; esiste infatti una regola fondamentale, la quale afferma che, per chi utilizza l'**Assembly**, il miglior debugger e' sicuramente il proprio cervello!
Conoscere l'**Assembly**, significa avere la capacita' di poter decifrare il codice macchina presente nei programmi eseguibili; proprio per questo motivo, i programmatori **Assembly** preferiscono utilizzare i debugger, per cercare di studiare il funzionamento di programmi eseguibili, per i quali non si dispone del codice sorgente.

Vediamo un esempio riferito sempre al programma **EXETEST.EXE**, che abbiamo creato nel precedente capitolo; proviamo ad eseguire il solito comando:
`..\bin\td exetest.exe`
In assenza delle informazioni di debugging, **TD** mostra una finestra con un messaggio del tipo:
`Program has no symbol table`
Questo messaggio, ci informa sul fatto che per **EXETEST.EXE**, non sono disponibili le informazioni di debugging, e quindi sara' possibile effettuare solamente un debugging di tipo "grezzo"; dopo aver chiuso la finestra di avvertimento, ci troviamo davanti alla situazione mostrata in Figura 8.

Figura 8 - Debugging di EXETEST.EXE

```
cs:0008 > B8DC11         mov ax,11DC                | eax 00000000 | c=0
cs:000B   8ED8           mov ds, ax                 | ebx 00000000 | z=0
cs:000D   C7062900AC0D   mov word ptr [0029],0DAC | ecx 00000000 | s=0
cs:0013   C7062B00401F   mov word ptr [002B],1F40 | edx 00000000 | o=0
cs:0019   C7062D00C21A   mov word ptr [002D],1AC2 | esi 00000000 | p=0
cs:001F   C7062F00E425   mov word ptr [002F],25E4 | edi 00000000 | a=0
cs:0025   C7063100F23C   mov word ptr [0031],3CF2 | ebp 00000000 | i=1
cs:002B   C7063300AB8D   mov word ptr [0033],8DAB | esp 00000400 | d=0
cs:0031   C70635002B5C   mov word ptr [0035],5C2B |  ds 11CC     |
cs:0037   C7063700F166   mov word ptr [0037],66F1 |  es 11CC     |
cs:003D   C7063900F193   mov word ptr [0039],93F1 |  fs 0000     |
cs:0043   C7063B009C35   mov word ptr [003B],359C |  gs 0000     |
cs:0049   C7063D00FF86   mov word ptr [003D],86FF |  ss 1235     |
cs:004F   C7063F00BC12   mov word ptr [003F],12BC |  cs 1220     |
cs:0055   C7068100BE0D   mov word ptr [0081],0DBE |  ip 0008     |
--------------------------------------------------+--------------+-----
ds:0100   0B CD AB 78 56 34 12 F0                 |  ss:0402   C435
ds:0108   DE BC 9A 78 56 34 12 44                 |  ss:0400 > C9A7
ds:0110   44 33 33 22 22 11 11 00                 |  ss:03FE   0000
ds:0108   00 C3 F5 48 40 A8 F0 E7                 |  ss:03FC   0F23
ds:0120   20 BD 5F 8F 96 D1 37 06                 |  ss:03FA   0A95
```

Osserviamo subito che **TD**, ci mostra il contenuto dei tre blocchi puntati inizialmente da **CS** (in alto a sinistra), **DS** (in basso a sinistra) e **SS** (in basso a destra); viene anche mostrato il contenuto iniziale di tutti i registri della **CPU**, compreso **FLAGS** (si tenga presente che le informazioni mostrate in Figura 8, possono variare da computer a computer).
Notiamo ora che:
`DS = ES = 11CCh`
Cio' significa che **EXETEST.EXE**, e' stato caricato in memoria a partire dall'indirizzo logico **11CCh:0000h**; questo e' anche l'indirizzo iniziale del **program segment**, per cui, possiamo dire che:
`StartSeg = PSP = DS = ES = 11CCh`
All'indirizzo logico **11CCh:0000h** corrisponde l'indirizzo fisico **11CC0h**; il **PSP** occupa **0100h** byte, per cui, il blocco **DATASEGM** parte dall'indirizzo fisico:
`11CC0h + 0100h = 11DC0h`
A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **11DCh:0000h**.
Dal **listing file** della Figura 2 del precedente capitolo, ricaviamo che il blocco **DATASEGM** occupa **0447h** byte; inoltre, il linker ha posto alla fine di **DATASEGM**, un buco di memoria da **1** byte per allineare **CODESEGM** alla **DWORD**. Cio' significa che **CODESEGM**, parte dall'indirizzo fisico:
`11DC0h + 0447h + 1h = 12208h`

A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **1220h:0008h**, che coincide anche con l'indirizzo dell'**entry point** del programma; infatti, in Figura 8 notiamo che appena **EXETEST.EXE** viene caricato in memoria, si ha:
`CS = 1220h` e `IP = 0008h`
Dal **listing file** della Figura 2 del precedente capitolo, ricaviamo che il blocco **CODESEGM** occupa **0147h** byte; inoltre, il linker ha posto alla fine di **CODESEGM**, un buco di memoria da **1** byte per allineare **STACKSEGM** al paragrafo. Cio' significa che **STACKSEGM**, parte dall'indirizzo fisico:
`12208h + 0147h + 1h = 12350h`
A questo indirizzo fisico, corrisponde l'indirizzo logico normalizzato **1235h:0000h**; inoltre, la dimensione iniziale dello stack e' pari a **0400h** byte. Infatti, in Figura 8 notiamo che appena **EXETEST.EXE** viene caricato in memoria, si ha:
`SS = 1235h` e `SP = 0400h`

Inizialmente, **DS** punta al **PSP**; infatti, all'indirizzo logico **DS:0000h** del blocco dati di Figura 8, troviamo i due codici macchina **CDh**, **20h** che, come sappiamo, si riferiscono all'istruzione:
`INT 20h`
per la terminazione in "vecchio stile" dei programmi **DOS**.
Se vogliamo trovare i dati statici del nostro programma, dobbiamo avanzare all'offset **0100h**, in modo da scavalcare i **256** byte del **PSP**; infatti, come si nota in Figura 8, all'indirizzo **DS:0100h** iniziano proprio i dati statici di **EXETEST.EXE**.

Nel blocco stack mostrato in Figura 8 in basso a destra, e' presente una freccia con la quale **TD** ci indica che l'indirizzo logico iniziale del **TOS** e' **SS:0400h**; come si puo' notare, lo stack e' pieno di dati il cui valore e' diverso da **0**. Si tratta di una diretta conseguenza del fatto che per le variabili temporanee, abbiamo richiesto **0400h** byte non inizializzati; di conseguenza, il blocco stack e' inizialmente pieno di dati casuali che, in gergo, vengono definiti **"spazzatura"**.

Nel blocco codice mostrato in Figura 8 in alto a sinistra, e' presente una freccia con la quale **TD** ci indica che l'indirizzo logico dell'**entry point** e' **CS:0008h**; la prima istruzione che viene eseguita e' quindi:
`mov ax, 11DCh`
Come abbiamo visto in precedenza, **11DCh** e' la componente **Seg** dell'indirizzo logico iniziale (**11DCh:0000h**) di **DATASEGM**; di conseguenza, la successiva istruzione:
`mov ds, ax`
carica **11DCh** in **DS**. Da questo momento, **DS:0000h** rappresenta l'indirizzo logico iniziale del blocco dati.
Consideriamo ora la terza istruzione:
`mov word ptr [0029h], 0DACh`
In base al **listing file** della Figura 2 del precedente capitolo, questa istruzione corrisponde a:
`mov varRect.p1.x, 3500`
Infatti, sempre dal **listing file**, ricaviamo che **varRect.p1.x** e' un dato a **16** bit, che si trova all'offset **0029h** del blocco **DATASEGM**.
In assenza delle informazioni di debugging, **TD** non e' ovviamente in grado di visualizzare i nomi simbolici che abbiamo assegnato ai dati del nostro programma; di conseguenza, nel caso della precedente istruzione, viene visualizzato l'indirizzo del dato (**DS:[0029h]**) e la sua dimensione in bit (attraverso l'operatore **WORD PTR**).

14.2.1 Esecuzione a passo singolo di un programma

Una delle caratteristiche piu' interessanti dei debugger come **TD**, e' quella di poter eseguire, a richiesta, solamente la prossima istruzione di un programma; questo metodo di debugging, prende il nome di **single-step mode** (modo di esecuzione a passo singolo). Nel caso di **TD**, si puo' ottenere l'esecuzione **single-step**, premendo ripetutamente il tasto funzione **F8**; in questo modo, e' possibile seguire "al rallentatore", l'evoluzione del programma in esecuzione.
Generalmente, questa tecnica si basa sull'utilizzo del **Trap Flag** presente nel registro **FLAGS**; come abbiamo visto in un precedente capitolo, se poniamo **TF=1**, la **CPU** genera una **INT 01h** dopo aver elaborato ogni singola istruzione del programma in esecuzione. Come conseguenza di questa **INT**, viene chiamata la **ISR** il cui indirizzo **Seg:Offset** e' memorizzato all'indice **01h** del vettore delle interruzioni; i debuggers, non fanno altro che installare una loro **ISR** per la gestione della **INT 01h**.
Ogni volta che la **CPU** elabora una istruzione del programma in esecuzione, genera una **INT 01h**; questa **INT** viene intercettata dalla **ISR** del debugger, che puo' cosi' visualizzare sullo schermo, tutti i dettagli relativi al programma stesso.
Come si puo' facilmente intuire, questa tecnica risulta essere particolarmente vulnerabile; un programma che non vuole farsi "debuggare", non deve fare altro che installare, a sua volta, una propria **ISR** per la gestione della **INT 01h**. Questa **ISR**, intercetta la **INT 01h** generata dalla **CPU** e, attraverso apposite istruzioni "maligne", provoca intenzionalmente un crash; si tratta di un semplice espediente, che viene utilizzato da molti programmi dotati di protezione "anti-debugger".

14.2.2 I breakpoint

Un'altra importante caratteristica dei debuggers, consiste nella possibilita' di eseguire un programma a velocita' normale, da una istruzione di partenza sino ad una istruzione di arrivo; quando l'esecuzione giunge alla istruzione di arrivo, il programma viene interrotto, dando cosi' la possibilita' al debugger, di visualizzare il contenuto di tutti i registri della **CPU**, e dei vari dati del programma stesso. Questa caratteristica risulta molto utile nel momento in cui, il programmatore, ritiene di aver individuato la "zona" di un programma, nella quale si verifica un malfunzionamento; il punto (scelto dal programmatore) nel quale il programma viene interrotto, prende il nome di **breakpoint** (punto di stop).
Come accade per l'esecuzione in **single-step mode**, anche la gestione dei **breakpoint** avviene attraverso un apposito vettore di interruzione; in questo caso, si tratta della **INT 03h**. A differenza delle altre **INT**, che hanno un codice macchina da **2** byte, la **INT 03h** ha un codice macchina (**CCh**) formato da un solo byte; in questo modo, si semplifica notevolmente il lavoro che il debugger deve svolgere per attivare o disattivare i **breakpoint** in un programma.
Per gestire i **breakpoint**, il debugger installa una propria **ISR** per la **INT 03h**; in fase di esecuzione di un programma, quando la **CPU** incontra il codice macchina **CCh**, genera una **INT 03h** che viene intercettata dalla **ISR** del debugger, che puo' cosi' procedere con le necessarie verifiche sul programma stesso.

Tutte le **CPU** della famiglia **80x86**, a partire dalla **80386**, sono state dotate di appositi registri interni, destinati esplicitamente ai debuggers; tutti questi argomenti, verranno trattati nelle sezioni **Assembly Avanzato** e **Modalita' Protetta**.

14.3 Il debugger del DOS

In mancanza di un debugger professionale, puo' rivelarsi molto utile anche il "mini-debugger" fornito dal **DOS**; si tratta del programma **DEBUG.EXE** presente nella cartella **C:\DOS** (nel caso di **Windows**, la cartella e' **C:\WINDOWS\COMMAND**).

La principale limitazione di **DEBUG**, e' rappresentata dal fatto che si tratta di un debugger a **16** bit; di conseguenza, questo debugger lavora in modo corretto, solo con programmi a **16** bit, contenenti esclusivamente istruzioni **8086** e **8087**.

Il programma **DEBUG** puo' essere eseguito in due modalita' distinte; la prima modalita', consiste nell'impartire dal prompt, il comando:
`debug`
Dopo aver premuto il tasto **[Invio]**, ci ritroviamo all'interno del debugger; sul bordo sinistro dello schermo, viene mostrato un trattino, attraverso il quale **DEBUG** ci informa che e' pronto a ricevere comandi.
Quando **DEBUG** viene lanciato in questo modo (cioe', senza nessun argomento), predispone automaticamente un **program segment** da **64** Kb che, nei primi **256** byte, contiene un generico **PSP**; i registri di segmento vengono inizializzati in questo modo:
`CS = DS = ES = SS = PSP`
I registri **IP** e **SP** vengono inizializzati in questo modo:
`IP = 0100h` e `SP = FFEEh`
Se si eccettua quindi **SP** che vale **FFEEh**, tutti gli altri registri vengono inizializzati secondo lo standard dei programmi in formato **COM**.
All'interno del **program segment** cosi' predisposto, e' possibile fare esperimenti di ogni genere; piu' avanti verranno illustrati degli esempi pratici.

Il secondo metodo per l'esecuzione di **DEBUG**, consiste nel passargli un file eseguibile come argomento; possiamo impartire, ad esempio, il comando:
`debug exetest.exe`
In un caso del genere, i vari registri vengono inizializzati secondo lo schema tipico degli eseguibili in formato **EXE**; nel nostro caso, per i registri di segmento **DS** e **ES** si ha:
`DS = ES = PSP`
La coppia **CS:IP** punta all'**entry point** da noi stabilito; la coppia **SS:SP** punta alla cima dello stack da noi predisposto.
Se invece impartiamo il comando:
`debug comtest.com`
i vari registri vengono inizializzati secondo lo schema tipico degli eseguibili in formato **COM**; nel nostro caso, per i registri di segmento si ha:
`CS = DS = ES = SS = PSP`
Per i registri **IP** e **SP** si ha:
`IP = 0100h` e `SP = FFFEh`

14.3.1 I comandi di DEBUG

Una volta che abbiamo lanciato il debugger, proviamo ad impartire il comando:
```
-?
```
(il trattino rappresenta il prompt di **debug**)
Premendo ora il tasto **[Invio]**, otteniamo una lista dei comandi che **debug** e' in grado di accettare; esaminiamo il significato dei vari comandi, con particolare riferimento a quelli piu' interessanti. Nei vari esempi che verranno illustrati, si suppone che **DEBUG** sia stato lanciato senza nessun argomento; per ogni comando, gli eventuali argomenti specificati tra parentesi quadre, sono facoltativi. Per maggiori dettagli, si consiglia di consultare i manuali del **DOS**.

***** Comando **Assemble**; sintassi: **A [indirizzo]**

Questo comando, ci permette di inserire direttamente in memoria, istruzioni **Assembly** appartenenti al set della **CPU 8086** e della **FPU 8087**; se impartiamo questo comando senza argomenti, **DEBUG** ci fornisce l'indirizzo logico iniziale, rappresentato da **CS:IP**. Supponendo, ad esempio, che **CS** contenga il valore **1331h**, e che **IP** contenga il valore **0100h**, in seguito al comando:
```
-a
```
il debugger ci mostra l'indirizzo logico:
```
1331:0100
```
Sulla destra di questo indirizzo, possiamo inserire l'istruzione **Assembly** desiderata; si noti che tutti gli indirizzi logici e tutti i valori immediati contenuti nelle istruzioni, devono essere espressi implicitamente in esadecimale.
In alternativa, possiamo specificare anche un indirizzo, attraverso il comando:
```
-a 1331:02ca
```
oppure:
```
-a cs:02ca
```
In questo caso, il debugger risponde mostrando la riga:
```
1331:02ca
```

A questo punto, possiamo procedere con l'inserimento delle istruzioni **Assembly**; sono disponibili le direttive **DB** e **DW**, gli address size operators **BYTE PTR** e **WORD PTR**, i commenti su una sola linea (**;**), i segment override, etc.
Ogni volta che inseriamo una nuova istruzione, **DEBUG** incrementa l'offset nel blocco codice, e ci mostra la prossima coppia libera **CS:Offset**; per terminare l'inserimento delle istruzioni, basta premere il tasto **[Invio]** in una riga vuota.

Vediamo un esempio pratico, che consiste nella visualizzazione di una stringa attraverso il servizio n. **09h** (**Display String**) della **INT 21h** (che e' stato gia' illustrato in precedenza); la sequenza delle istruzioni da inserire, viene mostrata in Figura 9.

Figura 9 - Mini-programma Assembly
```
-a
1331:0100 db   "Stringa di prova", "$"
1331:0111 mov  dx, 0100       ; dx = offset stringa
1331:0114 mov  ah, 09         ; servizio Display String
1331:0116 int  21             ; chiama i servizi DOS
1331:0118 mov  ah, 4c         ; servizio Terminate Program
1331:011A mov  al, 00         ; exit code = 0
1331:011C int  21             ; chiama i servizi DOS
1331:011E
```

Anche in questo caso, **DS** non necessita di nessuna modifica, in quanto contiene gia' la componente **Seg** dell'indirizzo logico della stringa; infatti, abbiamo appena visto che **DEBUG**, lanciato senza argomenti, crea un unico segmento di programma che parte da un indirizzo logico la cui componente **Seg** viene caricata in **CS**, **DS**, **ES** e **SS**.
In **DX**, viene caricato il valore **0100h**, che e' l'offset della stringa da visualizzare; possiamo maneggiare tranquillamente questi indirizzi assoluti, in quanto stiamo inserendo le istruzioni direttamente in memoria (non esiste quindi nessuna rilocazione degli indirizzi).
Per eseguire questo programma, possiamo servirci del comando **G** (**Go**), che viene descritto piu' avanti; nel nostro caso, possiamo notare che il codice eseguibile inizia da **CS:0111h**. Il comando da impartire e' quindi:

```
-g = 1331:0111
```

oppure:

```
-g = cs:0111
```

In questo modo, **DEBUG** esegue il programma che, dopo aver visualizzato la stringa, restituisce il controllo al **DOS**.

E' importante che le varie istruzioni **Assembly**, vengano inserite nel **program segment** predisposto da **DEBUG**; in caso contrario, si corre il rischio di sovrascrivere aree riservate della memoria, con conseguente crash del programma.

***** Comando **Compare**; sintassi: **C intervallo indirizzo**

Con questo comando, e' possibile comparare due aree di memoria, in modo da evidenziare eventuali differenze nel loro contenuto; il debugger visualizza esclusivamente i byte che, nelle due aree da confrontare, occupano la stessa posizione, e differiscono tra loro in valore. Il formato di visualizzazione e' del tipo:

```
indirizzo1 byte1 byte2 indirizzo2
```

L'argomento **intervallo** e' formato dall'indirizzo iniziale del primo blocco, e dall'offset finale del blocco stesso; l'argomento **indirizzo** rappresenta l'indirizzo iniziale del secondo blocco.
Consideriamo il seguente esempio:

```
-c ds:0f00 0f10 ss:0200
```

Questo comando richiede la comparazione tra i primi **11h** byte (da **DS:0F00h** a **DS:0F10h**) del blocco di memoria che inizia da **DS:0F00h**, e i primi **11h** byte del blocco di memoria che inizia da **SS:0200h**; se, ad esempio, il terzo byte del primo blocco vale **2Fh**, e il terzo byte del secondo blocco vale **1Ah**, verra' mostrato l'output:

```
DS:0F02 2F 1A SS:0202
```

***** Comando **Dump**; sintassi: **D [intervallo]**

Con questo comando, e' possibile visualizzare il contenuto di un determinato blocco di memoria (**memory dump**); in assenza di argomenti, vengono visualizzati i primi **128** byte del blocco di memoria che inizia dall'**entry point**.
In alternativa, e' possibile specificare l'indirizzo iniziale e l'offset finale del blocco da visualizzare; se, ad esempio, vogliamo visualizzare i primi **21h** byte del blocco di memoria che parte dall'indirizzo **SS:FB00h**, dobbiamo impartire il comando:

```
-d ss:fb00 fb20
```

Vediamo un esempio pratico che ci permette di effettuare il **dump** dell'area della **RAM** riservata ai vettori di interruzione; in base a quanto e' stato esposto nei precedenti capitoli, sappiamo che quest'area parte dall'indirizzo logico **0000h:0000h**, e contiene **256** coppie **Seg:Offset**. La Figura 10, mostra il blocco dei primi **8** vettori di interruzione; questo blocco, inizia quindi dall'indirizzo logico **0000h:0000h**, e termina all'offset **001Fh** (**4*8=20h** byte).

Figura 10 - Dump della memoria

```
-d 0000:0000 001f
0000:0000 8A 10 16 01 00 04 70 00 - 16 00 72 D5 65 04 70 00  ......p...r.e.p.
0000:0010 65 04 70 00 54 FF 00 F0 - 1F 94 00 F0 53 FF 00 F0  e.p.T.......S...
```

Come si puo' notare, nella parte destra **DEBUG** visualizza anche i codici ASCII associati ai vari byte presenti in memoria (al posto dei codici minori di **20h** e maggiori di **7Eh**, viene visualizzato un punto); si tenga presente che le informazioni mostrate in Figura 10, possono variare da computer a computer, e da versione a versione del **DOS** o di **Windows**.
Per interpretare correttamente le informazioni presenti in Figura 10, e' necessario tenere presente che tutte le **CPU** della famiglia **80x86**, seguono una importante convenzione per la memorizzazione degli indirizzi logici **Seg:Offset**; questa convenzione prevede che la componente **Offset**, preceda in memoria la componente **Seg**.
E' anche importante osservare che **DEBUG**, mostra la memoria con gli indirizzi crescenti da sinistra verso destra; come gia' sappiamo, in un caso del genere i dati di tipo **WORD**, **DWORD**, etc, appaiono disposti al contrario rispetto alla loro rappresentazione con il sistema posizionale.

Sulla base di tutte queste considerazioni, dalla Figura 10 possiamo ricavare gli indirizzi **Seg:Offset**, da cui partono in memoria le **ISR** che gestiscono i vari vettori di interruzione; come e' stato gia' spiegato in un precedente capitolo, queste **ISR** sono destinate esclusivamente ai programmi che lavorano in modalita' reale. A titolo di curiosita', esaminiamo i primi **4** vettori.

Sul **PC** a cui si riferisce la Figura 10, la **ISR** predefinita per la gestione della **INT 00h**, si trova in memoria all'indirizzo logico **0116h:108Ah**; si tratta di una **INT** hardware, che viene generata dalla **CPU** ogni volta che in un programma, si verifica una divisione per zero (**overflow di divisione**). In fase di avvio del computer, il **DOS** installa una propria **ISR** per la gestione di questa **INT**; a loro volta, anche i normali programmi possono installare una apposita **ISR** per la **INT 00h**.

Sul **PC** a cui si riferisce la Figura 10, la **ISR** predefinita per la gestione della **INT 01h**, si trova in memoria all'indirizzo logico **0070h:0400h**; come e' stato gia' spiegato in questo capitolo, se poniamo **TF=1** nel registro **FLAGS**, la **CPU** genera una **INT 01h** (hardware) dopo l'elaborazione di ogni istruzione del programma in esecuzione. In questo modo, i debugger possono eseguire i programmi in **single-step mode**; anche i normali programmi possono installare una apposita **ISR** per la **INT 01h**.

Sul **PC** a cui si riferisce la Figura 10, la **ISR** predefinita per la gestione della **INT 02h**, si trova in memoria all'indirizzo logico **D572h:0016h**; questa **INT** ha la massima priorita' possibile, in quanto viene generata dall'hardware del computer, per segnalare l'insorgere di un grave problema interno. Data la sua importanza, la **INT 02h** non puo' essere inibita (mascherata) attraverso l'istruzione **CLI** (**clear interrupt enable flag**); proprio per questo motivo, questa **INT** viene chiamata **NMI** o **Non Maskable Interrupt** (interruzione non mascherabile). La **INT 02h** arriva direttamente alla **CPU** attraverso un apposito piedino chiamato **NMI**; vedere a tale proposito la Figura 3 e la Figura 6 del Capitolo 9.

Sul **PC** a cui si riferisce la Figura 10, la **ISR** predefinita per la gestione della **INT 03h**, si trova in memoria all'indirizzo logico **0070h:0465h**; come e' stato gia' spiegato in questo capitolo, la **INT 03h** viene generata dal codice macchina **CCh** (interruzione software), inserito dai debugger per la gestione dei **breakpoint**.

Tutti questi argomenti, verranno trattati in dettaglio nella sezione **Assembly Avanzato**.

***** Comando **Enter**; sintassi: **E indirizzo [elenco]**

Questo comando, permette di inserire dei byte, a partire dall'indirizzo di memoria specificato dall'argomento **indirizzo**; e' anche possibile specificare un **elenco** di byte da inserire in sequenza, a partire dalla posizione **indirizzo**.
Ad esempio, il seguente comando:
```
-e cs:0100 3a 2b 1f ff 22 4d 5e 6f
```
inserisce in memoria **8** byte, a partire dall'indirizzo logico **CS:0100h**.

***** Comando **Fill**; sintassi: **F intervallo elenco**

Questo comando, inserisce un **elenco** di byte, all'interno del blocco di memoria specificato da **intervallo**; ad esempio, il seguente comando:
```
-f ds:00d0 00d3 a2 b4 cd f9
```
inserisce **4** byte, nel blocco di memoria compreso tra gli indirizzi logici **DS:00D0h** e **DS:00D3h**.

***** Comando **Go**; sintassi: **G [=indirizzo] [indirizzi]**

Questo comando, permette di eseguire un programma che si trova in memoria; in assenza di argomenti, il comando **Go** avvia l'esecuzione a partire dall'**entry point**.
In alternativa, il programma puo' essere avviato anche a partire da **=indirizzo**; l'altro argomento **indirizzi**, permette di specificare uno o piu' **breakpoint**.
Ad esempio, nel caso del programma di Figura 9, il comando:
```
-g = cs:0111 cs:0118
```
provoca l'esecuzione delle prime tre istruzioni; all'indirizzo logico **CS:0118h**, il debugger salva il primo byte del codice macchina dell'istruzione:
```
mov ah, 4ch
```
e lo sostituisce con il codice macchina **CCh**, che provoca, come sappiamo, la generazione di una **INT 03h**. Subito dopo l'interruzione del programma, il debugger ripristina il vecchio codice macchina presente all'indirizzo **CS:0118h**, e visualizza lo stato di tutti i registri della **CPU**, compreso **FLAGS**.

***** Comando **Hex**; sintassi: **H valore1 valore2**

Questo comando, permette di effettuare somme e sottrazioni tra valori esadecimali a **8** o **16** bit; ad esempio, il comando:
```
-h 4fff 003b
```
produce il seguente output:
```
503A 4FC4
```
Il primo valore, rappresenta la somma tra **4FFFh** e **003Bh**; il secondo valore, rappresenta la differenza tra **4FFFh** e **003Bh**.

***** Comando **Input**; sintassi: **I porta**

Questo comando, visualizza un byte letto da una porta hardware del **PC**; l'indirizzo della porta, viene specificato attraverso l'argomento **porta**.
Consideriamo, ad esempio, la porta hardware n. **60h**, che rappresenta l'indirizzo del buffer nel quale viene memorizzata la coda degli scan codes, relativi ai tasti premuti o rilasciati nella tastiera; per leggere il codice che si trova in cima alla coda, possiamo scrivere:
```
-i 60
```
Chi ha a disposizione un normale joystick a due assi e due pulsanti, puo' anche provare a leggere

l'input dalla porta joystick standard **0201h**; a tale proposito, bisogna impartire il comando:

```
-i 201
```

Tenendo premuti, nessuno, uno o entrambi i pulsanti, si possono vedere le variazioni del valore letto dalla porta joystick.

Si consiglia di non assegnare numeri a caso all'argomento **porta**; infatti, la lettura di particolari porte hardware, puo' provocare un blocco del **PC**.
Sui **SO** come **Windows XP**, viene fornito un emulatore **DOS** con diverse limitazioni rispetto al **DOS** vero e proprio; queste limitazioni riguardano, in particolare, l'accesso diretto alle porte hardware da parte dei programmi **DOS**.
Tutto cio' che riguarda l'**I/O** con le porte hardware, verra' trattato nella sezione **Assembly Avanzato**.

***** Comando **Load**; sintassi: **L [indirizzo] [unita'] [primosettore] [numero]**

Questo comando, carica dal disco, un file precedentemente passato come argomento a **debug**, o un file precedentemente nominato con il comando **Name** (illustrato piu' avanti); l'argomento **indirizzo** permette di specificare l'indirizzo logico di memoria da cui iniziera' il caricamento del programma.
Gli altri parametri, fanno riferimento a settori logici del disco specificato da **unita'** (**0=A:**, **1=B:**, etc).
Nota Importante.
In seguito alla formattazione, un disco (floppy disk o hard disk) viene suddiviso in tanti cerchi concentrici, chiamati **tracce**; le varie tracce, a partire da quella piu' esterna, vengono individuate dagli indici **0**, **1**, **2**, etc. Ogni singola traccia, viene suddivisa in tante parti, chiamate **settori**; i vari settori di ogni traccia, vengono individuati dagli indici **1**, **2**, **3**, etc. Il sistema di suddivisione appena descritto, permette di accedere rapidamente ad una qualsiasi area del disco; infatti, ogni area viene individuata univocamente da una coppia [**traccia**, **settore**].
Una volta che un disco e' stato formattato, e' possibile creare al suo interno, il cosiddetto **file system**; il file system e' un insieme di convenzioni, utilizzate dai **SO** per la gestione dei file su disco. Tra i file system piu' conosciuti, si possono citare, ad esempio, **FAT16** (usato dal **DOS**), **FAT32** (usato da **Windows 98**), **NTFS** (usato da **Windows XP**), **ext2** (usato da **Linux**), etc.
Nel caso generale, i file system dei **SO**, vedono un disco sotto forma di vettore lineare di elementi, chiamati **settori logici**; il driver che pilota il disco, ha il compito di convertire i settori logici in coordinate [**traccia**, **settore**].
Accedere ad un disco attraverso il metodo utilizzato dal comando **Load**, significa scavalcare il controllo che il **SO** ha sul file system; proprio per questo motivo, si raccomanda vivamente di evitare l'uso del comando **Load**.

***** Comando **Move**; sintassi: **M intervallo indirizzo**

Questo comando, permette di trasferire il contenuto di un blocco di memoria specificato da **intervallo**, in un'altra area della memoria che inizia da **indirizzo**; possiamo scrivere, ad esempio:

```
-m cs:0200 02ff cs:0400
```

In questo caso, i **100h** byte compresi tra gli indirizzi **CS:0200h** e **CS:02FFh**, vengono trasferiti in un'area di memoria da **100h** byte, che inizia dall'indirizzo **CS:0400h**; come al solito, si raccomanda vivamente di effettuare questo tipo di operazioni, solo all'interno dell'area di lavoro predisposta da **debug**.
Il comando **Move** lavora correttamente anche quando il blocco sorgente e il blocco destinazione sono parzialmente sovrapposti; la parte del blocco sorgente, destinata ad essere sovrascritta dal blocco destinazione, viene trasferita per prima.

***** Comando **Name**; sintassi: **N [nomepercorso] [elencoargomenti]**

Questo comando, permette di specificare uno o piu' file che possono essere caricati dall'interno di **debug**; il caricamento dei file avviene poi attraverso il comando **Load** (illustrato in precedenza).

***** Comando **Output**; sintassi: **O porta byte**

Questo comando, permette di scrivere un valore **byte** a **8** bit, nella porta hardware che si trova all'indirizzo **porta**; la scrittura di dati casuali in determinate porte hardware, puo' provocare un crash del **PC**. Come e' stato detto in precedenza, tutto cio' che riguarda le operazioni di **I/O** con le porte hardware, verra' trattato nella sezione **Assembly Avanzato**.

***** Comando **Quit**; sintassi: **Q**

Questo comando, termina l'esecuzione di **debug**, e restituisce il controllo al **DOS**.

***** Comando **Register**; sintassi: **R [registro]**

Questo comando, visualizza il contenuto di uno o piu' registri della **CPU**; in assenza dell'argomento **registro**, viene visualizzato il contenuto di tutti i registri della **CPU**, compreso **FLAGS**.
Impartendo, ad esempio, il comando:
```
-r ax
```
viene visualizzato il contenuto del solo registro **AX**; subito dopo, **debug** attende che l'utente inserisca un nuovo valore da assegnare ad **AX**.
Gli unici nomi validi per i registri sono, **AX**, **BX**, **CX**, **DX**, **SP**, **BP**, **SI**, **DI**, **CS**, **DS**, **ES**, **SS**, **IP** (o **PC**), **F** (registro dei **FLAGS**); come si puo' notare, **debug** permette di modificare anche il contenuto dell'**instruction pointer** (o **program counter**).
Per modificare uno o piu' flags, bisogna quindi impartire il comando:
```
-r f
```
Il debugger risponde mostrando una linea del tipo:
```
NV UP EI PL NZ NA PO NC -
```
Il trattino finale indica che **debug** resta in attesa che l'utente inserisca i nuovi valori da assegnare a uno o piu' flags; le modifiche diventano effettive, subito dopo la pressione del tasto **[Invio]**. La sintassi utilizzata per i soli **8** flags disponibili, viene mostrata in Figura 11.

Figura 11 - Flags

FLAG	SET	CLEAR
OF	OV	NV
DF	DN (decrement)	UP (increment)
IF	EI (enabled)	DI (disabled)
SF	NG (negative)	PL (plus)
ZF	ZR	NZ
AF	AC	NA
PF	PE (even)	PO (odd)
CF	CY	NC

In questa tabella, la colonna **SET** si riferisce ai flags a livello logico **1**; la colonna **CLEAR** si riferisce ai flags a livello logico **0**.

***** Comando **Search**; sintassi: **S intervallo elenco**

Questo comando, permette di cercare in memoria, una sequenza formata da uno o piu' byte (consecutivi e contigui); l'argomento **intervallo** indica il blocco di memoria nel quale effettuare la ricerca, mentre l'argomento **elenco** indica la sequenza da cercare.
Impartendo, ad esempio, il comando:
```
-s cs:0300 04ff 3d 2a b2
```
stiamo richiedendo la ricerca della sequenza **3Dh**, **2Ah**, **B2h**, in un blocco di memoria da **200h** byte, compreso tra gli indirizzi **CS:0300h** e **CS:04FFh**; il debugger visualizza, eventualmente, tutti gli indirizzi di questo blocco, a partire dai quali e' presente la sequenza cercata.

***** Comando **Trace**; sintassi: **T [=indirizzo] [valore]**

Questo comando, permette di eseguire un programma in **single-step mode**; in assenza di parametri, viene eseguita per prima l'istruzione puntata da **CS:IP**.
E' anche possibile richiedere l'esecuzione della sola istruzione che si trova in posizione **indirizzo**; l'argomento opzionale **valore**, indica il numero di istruzioni da eseguire.
Nel caso, ad esempio, del programma di Figura 9, possiamo eseguire la sola istruzione:
```
mov ah, 4ch
```
attraverso il comando:
```
-t = 1331:0118
```
Terminata l'esecuzione dell'istruzione, **debug** visualizza lo stato di tutti i registri della **CPU**.

***** Comando **Unassemble**; sintassi: **U [intervallo]**

Questo comando, e' formalmente simile a **Dump**; la differenza sostanziale sta nel fatto che **Unassemble**, visualizza il contenuto della memoria sotto forma di istruzioni **Assembly**.
Impartendo, ad esempio, il comando:
```
-u cs:0200 020f
```
viene mostrato il disassemblaggio di un blocco di memoria da **10h** byte, compreso tra gli indirizzi logici **CS:0200h** e **CS:020Fh**; il listato **Assembly** che si ottiene, potrebbe essere totalmente privo di senso!
Bisogna tenere presente che **Unassemble**, cerca di convertire in istruzioni **Assembly** tutto cio' che incontra nel blocco di memoria specificato; se questo blocco contiene, ad esempio, dati statici, si ottengono ovviamente istruzioni **Assembly** sbagliate.
E' anche necessario ricordare che **debug**, lavora correttamente solo in presenza di istruzioni appartenenti al set della **CPU 8086** e della **FPU 8087**; in presenza di codici macchina a **32** bit, si ottengono istruzioni **Assembly** insensate.

***** Comando **Write**; sintassi: **W [indirizzo] [unita'] [primosettore] [numero]**

Questo comando (che e' la controparte di **Load**), permette di salvare su disco, il contenuto di un'area di memoria che parte da **indirizzo**; per accedere al disco, bisogna specificare, il numero di **unita'**, il settore logico iniziale (**primosettore**) e il **numero** dei blocchi da scrivere. Come si puo' facilmente intuire, questo metodo di accesso al disco permette di scavalcare il controllo che il **SO** ha sul file system; a differenza, pero', di quanto accade con il comando **Load** (lettura dal disco), il comando **Write** effettua una operazione di scrittura sul disco, che potrebbe rivelarsi dannosa!

Si raccomanda quindi di evitare nella maniera piu' assoluta, l'utilizzo del comando **Write**; la scrittura diretta su disco, puo' provocare danni irreparabili al file system, con conseguente perdita dei dati memorizzati!

14.4 Conclusioni

Sarebbe interessante a questo punto, poter scrivere dei programmi che, prendendo spunto dai debugger, visualizzano sullo schermo, in tempo reale, tutte le informazioni relative al proprio "assetto" in memoria; in questo modo, si potrebbe ottenere una ulteriore verifica pratica, dei concetti teorici illustrati nel precedente capitolo.
Un programma del genere, dovrebbe essere in grado di leggere il contenuto dei registri della **CPU**, e di mostrarlo sullo schermo; purtroppo pero', a differenza di quanto accade con i linguaggi di alto livello, l'**Assembly** non dispone di nessuna procedura predefinita per la gestione dell'input e dell'output delle informazioni.
Come e' stato detto in un precedente capitolo, l'**Assembly** piu' che un linguaggio di programmazione, e' un insieme di strumenti attraverso i quali si puo' fare di tutto; se vogliamo gestire l'input dalla tastiera o l'output sullo schermo, non dobbiamo fare altro che scrivere apposite procedure. La scrittura di queste procedure, richiede pero' la conoscenza dell'**Assembly**; si viene a creare quindi un paradosso, in base al quale, **per conoscere l'Assembly dobbiamo scrivere dei programmi, ma per scrivere dei programmi, dobbiamo gia' conoscere l'Assembly**!
Per poter uscire da questo vicolo cieco, l'unica possibilita' e' quella di utilizzare, inizialmente, una libreria di procedure, gia' scritte da altri programmatori; man mano che si impara a programmare in **Assembly**, si acquisiscono anche le conoscenze necessarie per scrivere in proprio queste procedure.
Nel prossimo capitolo, verra' utilizzata una di queste librerie, che ci permettera' di gestire in modo molto semplice, le operazioni di input dalla tastiera, e di output sullo schermo.

Capitolo 15 - Libreria di procedure per l'I/O

Tutti i linguaggi di alto livello, mettono a disposizione dei programmatori, sofisticati ambienti di sviluppo che comprendono, in particolare, una numerosa collezione di cosiddetti **sottoprogrammi**, "pronti per l'uso"; con il termine sottoprogramma, si indica un piccolo blocco di istruzioni che, nel loro insieme, formano una sorta di "mini-programma".
Ciascun sottoprogramma, e' destinato a svolgere un compito ben preciso; possiamo avere, ad esempio, sottoprogrammi che si occupano della gestione della memoria, della gestione dei file su disco, della lettura dell'input dalla tastiera, della scrittura dell'output sullo schermo e cosi' via.
I sottoprogrammi possono essere collegati, direttamente o indirettamente, ad un generico programma che possiamo definire **programma principale**; ogni volta che il programma principale deve svolgere un determinato compito, non deve fare altro che chiamare l'opportuno sottoprogramma.
Si ha il collegamento diretto, quando tutti i sottoprogrammi di cui abbiamo bisogno, vengono inseriti nello stesso modulo contenente anche il programma principale; esiste pero' un sistema molto piu' pratico, rappresentato dal collegamento indiretto. Prima di tutto, in base al compito svolto, i vari sottoprogrammi vengono suddivisi in gruppi, e ciascun gruppo, viene inserito in un file distinto; ciascuno dei file cosi' ottenuti, prende il nome di **libreria di sottoprogrammi**. Ogni volta che scriviamo un programma che ha bisogno di una serie di sottoprogrammi, non dobbiamo fare altro che linkare le opportune librerie, al modulo contenente il programma principale.

Nei linguaggi di alto livello, per indicare i diversi tipi di sottoprogrammi, vengono utilizzati termini come, **procedura**, **funzione**, **subroutine**; in **Assembly**, i sottoprogrammi vengono chiamati **procedure**. Nel seguito del capitolo, e in tutta la sezione **Assembly Base**, verra' sempre utilizzato il termine **procedura** per indicare un generico sottoprogramma; le procedure **Assembly**, verranno trattate in dettaglio in un apposito capitolo.

Come e' stato detto in precedenza, gli utenti dei linguaggi di alto livello, hanno a disposizione numerose librerie di procedure pronte per l'uso, che possono essere utilizzate "a scatola chiusa", senza preoccuparsi di come esse svolgano il proprio lavoro; del resto, uno degli scopi fondamentali dei linguaggi di alto livello, e' proprio quello di consentire a chiunque di scrivere programmi, senza la necessita' di conoscere il funzionamento interno del computer.
Per raggiungere questo obiettivo, i linguaggi di alto livello vengono strutturati in modo da rendere la programmazione, indipendente dall'hardware del computer; il compilatore o l'interprete, si occupano di tutti i dettagli relativi alla traduzione del codice sorgente, nel codice macchina comprensibile dalla piattaforma hardware che si sta utilizzando.
I linguaggi di alto livello, rappresentano quindi la soluzione ideale per quei programmatori che non vogliono perdere tempo ad armeggiare, direttamente, con **CPU**, **FPU**, **Bus PCI**, **Bus AGP**, interfacce **IDE**, **EIDE**, **SCSI**, etc; le comodita' offerte dai linguaggi di alto livello, si pagano pero' a caro prezzo in termini di dimensioni e prestazioni del codice prodotto. E' chiaro, infatti, che piu' la struttura di un linguaggio si allontana dall'hardware del computer, piu' diventa complesso e contorto il lavoro di traduzione del codice sorgente in codice macchina; tutto cio' porta ad ottenere programmi eseguibili di dimensioni spropositate, e con prestazioni insoddisfacenti.

I linguaggi di alto livello, mostrano tutti i loro limiti nel momento in cui si presenta la necessita' di scrivere programmi potenti ed efficienti, capaci di sfruttare al massimo l'hardware che il computer mette a disposizione; i programmatori alle prese con i vari **Java**, **VisualBasic**, **Delphi**, etc, si trovano davanti ad una vera e propria barriera protettiva, che impedisce loro di accedere in modo efficiente, all'hardware sottostante. Se si vuole scavalcare questa barriera, esiste una sola strada da seguire, e consiste nell'acquisire una conoscenza approfondita dell'architettura del computer; una volta che questa conoscenza e' stata acquisita, si e' in grado dialogare direttamente con il computer,

utilizzando il suo stesso linguaggio. Come gia' sappiamo, il computer parla il linguaggio binario, rappresentato da sequenze di **0** e **1** che, nel loro insieme, formano il codice macchina; per evitare ai programmatori di dover lavorare direttamente con il codice macchina, e' stato creato il linguaggio **Assembly**.

A differenza di quanto accade con i linguaggi di alto livello, l'**Assembly** non mette a disposizione nessuna procedura predefinita da utilizzare a scatola chiusa per gestire la memoria, i file su disco, l'input dalla tastiera, l'output sullo schermo, etc; tutto il lavoro necessario per realizzare queste procedure, viene lasciato al programmatore. Tutto cio' che l'**Assembly** mette a disposizione, e' rappresentato dal set di istruzioni della **CPU**; mettendo assieme queste semplici istruzioni, e' possibile fare qualunque cosa, purche', ovviamente, si sappia quello che si sta facendo.
I programmatori **Assembly** sparsi in tutto il mondo, hanno provveduto a realizzare e a mettere in circolazione, un numero enorme di librerie di procedure **Assembly**, destinate a coprire praticamente tutti gli aspetti delle conoscenze umane; e' chiaro pero' che l'obiettivo fondamentale di chi programma in **Assembly**, non e' quello di utilizzare queste procedure a scatola chiusa, ma e' quello di capire come scrivere di persona le procedure di cui si ha bisogno.
Nel capitolo precedente, e' stato messo in evidenza il paradosso che si trova davanti chi vuole imparare a programmare in **Assembly**; se, ad esempio, si vuole stampare una stringa sullo schermo, bisogna scrivere una apposita procedura **Assembly**, ma per scrivere questa procedura, bisogna gia' conoscere l'**Assembly**. Per uscire da questa situazione, l'unica cosa da fare consiste nel servirsi, inizialmente, di apposite librerie di procedure gia' pronte per l'uso; man mano che si impara a programmare in **Assembly**, si acquisiscono le conoscenze necessarie per scrivere in proprio queste procedure.

15.1 Le librerie EXELIB e COMLIB

In questo capitolo, viene illustrata una libreria che ci permette di gestire in modo molto semplice, le operazioni per l'input dalla tastiera e per l'output sullo schermo; utilizzando le procedure presenti in questa libreria, possiamo verificare in pratica tutti i concetti esposti nei precedenti capitoli, e possiamo poi procedere in modo piu' rapido, nell'apprendimento del linguaggio **Assembly**.

La libreria viene fornita in due versioni, contenute nei due file zippati, chiamati **EXELIB.ZIP** e **COMLIB.ZIP**; questi due file, possono essere scaricati dalla pagina dei **Downloads** di questo sito (nella sezione **Software**).

Decomprimendo nella cartella **ASMBASE** il file **EXELIB.ZIP**, si ottengono i seguenti tre file:
EXELIB.OBJ, che contiene la libreria di procedure da linkare ai programmi in formato **DOS EXE**;
EXELIB.INC, che contiene le dichiarazioni delle varie procedure presenti in **EXELIB.OBJ**;
EXELIB.TXT, che contiene la descrizione delle varie procedure presenti in **EXELIB.OBJ**.

Analogamente, decomprimendo nella cartella **ASMBASE** il file **COMLIB.ZIP**, si ottengono i seguenti tre file:
COMLIB.OBJ, che contiene la libreria di procedure da linkare ai programmi in formato **DOS COM**;
COMLIB.INC che contiene le dichiarazioni delle varie procedure presenti in **COMLIB.OBJ**;
COMLIB.TXT, che contiene la descrizione delle varie procedure presenti in **COMLIB.OBJ**.

Sia **EXELIB.OBJ**, che **COMLIB.OBJ**, contengono le stesse identiche procedure; l'unica differenza, consiste nel fatto che, come e' stato gia' detto, la libreria **EXELIB.OBJ** e' destinata ad essere linkata ai programmi in formato **DOS EXE**, mentre la libreria **COMLIB.OBJ** e' destinata ad essere linkata ai programmi in formato **DOS COM**.

15.1.1 Parametri in ingresso e valore in uscita di una procedura

Una procedura, per poter svolgere il proprio lavoro, puo' richiedere una serie di informazioni, chiamate simbolicamente, **entry parameters** o **parametri in ingresso**; le informazioni effettive che il programmatore passa alla procedura, come parametri in ingresso, prendono il nome di **entry arguments** o **argomenti in ingresso**. Su alcuni libri, i parametri vengono anche chiamati **parametri formali**, mentre gli argomenti vengono anche chiamati **parametri attuali**; nella sezione **Assembly Base**, verra' sempre utilizzato il termine **parametri** per indicare la lista simbolica di informazioni richieste da una procedura, e il termine **argomenti** per indicare la lista effettiva di informazioni che il programmatore passa alla procedura stessa.
Alcune procedure, dopo aver svolto il proprio lavoro, restituiscono a chi le ha chiamate, una informazione che viene definita **exit value** o **valore in uscita**; spesso, il valore in uscita contiene il risultato delle elaborazioni a cui sono stati sottoposti gli argomenti ricevuti in ingresso dalla procedura.
Tutte le procedure contenute in **EXELIB.OBJ** e **COMLIB.OBJ**, utilizzano i registri della **CPU**, sia per ricevere gli eventuali argomenti in ingresso, sia per restituire un eventuale valore in uscita; come vedremo nei capitoli successivi, questo metodo, pur non essendo il piu' raffinato, e' sicuramente il piu' veloce.

15.1.2 Descrizione delle procedure presenti nella libreria

Esaminiamo ora in dettaglio, l'elenco delle procedure disponibili nella libreria, e il compito svolto da ciascuna di esse; nella descrizione che segue, per ogni procedura viene utilizzato il termine **Entry** per indicare la lista simbolica degli eventuali parametri in ingresso, e il termine **Exit** per indicare l'eventuale valore in uscita.

```
setCursorPosition: modifica la posizione del cursore lampeggiante sullo schermo.
   Entry: DH = riga (da 0 a 49)
          DL = colonna (da 0 a 79)
   Exit:  nessun valore in uscita

showCursor: rende visibile il cursore lampeggiante sullo schermo.
   Entry: nessun parametro in ingresso
   Exit:  nessun valore in uscita

hideCursor: rende invisibile il cursore lampeggiante sullo schermo.
   Entry: nessun parametro in ingresso
   Exit:  nessun valore in uscita

clearScreen: cancella lo schermo.
   Entry: nessun parametro in ingresso
   Exit:  nessun valore in uscita

set80x25: seleziona la modalita' video testo 80 colonne x 25 righe a 16 colori.
   Entry: nessun parametro in ingresso
   Exit:  nessun valore in uscita

set80x50: seleziona la modalita' video testo 80 colonne x 50 righe a 16 colori.
   Entry: nessun parametro in ingresso
   Exit:  nessun valore in uscita

waitChar: attende che venga premuto un tasto qualunque sulla tastiera.
   Entry: nessun parametro in ingresso
   Exit:  AL = codice ASCII del tasto premuto
          AH = ScanCode del tasto premuto
```

writeString: stampa una stringa C sullo schermo.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
EXELIB: ES:DI = indirizzo logico Seg:Offset della stringa da stampare
COMLIB: DI = componente Offset dell'indirizzo logico della stringa da stampare
Exit: nessun valore in uscita

writeUdec8: stampa sullo schermo un numero intero senza segno a 8 bit in base 10.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
AL = numero da stampare
Exit: nessun valore in uscita

writeUdec16: stampa sullo schermo un numero intero senza segno a 16 bit in base 10.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
AX = numero da stampare
Exit: nessun valore in uscita

writeUdec32: stampa sullo schermo un numero intero senza segno a 32 bit in base 10.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
EAX = numero da stampare
Exit: nessun valore in uscita

writeSdec8: stampa sullo schermo un numero intero con segno a 8 bit in base 10.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
AL = numero da stampare
Exit: nessun valore in uscita

writeSdec16: stampa sullo schermo un numero intero con segno a 16 bit in base 10.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
AX = numero da stampare
Exit: nessun valore in uscita

writeSdec32: stampa sullo schermo un numero intero con segno a 32 bit in base 10.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
EAX = numero da stampare
Exit: nessun valore in uscita

writeHex8: stampa sullo schermo un numero intero a 8 bit in base 16.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
AL = numero da stampare
Exit: nessun valore in uscita

writeHex16: stampa sullo schermo un numero intero a 16 bit in base 16.
Entry: DH = riga (da 0 a 49)
DL = colonna (da 0 a 79)
AX = numero da stampare
Exit: nessun valore in uscita

writeHex32: stampa sullo schermo un numero intero a 32 bit in base 16.
Entry: DH = riga (da 0 a 49)

```
         DL = colonna (da 0 a 79)
         EAX = numero da stampare
  Exit:  nessun valore in uscita
```

writeBin8: stampa sullo schermo un numero intero a 8 bit in base 2.

```
  Entry: DH = riga (da 0 a 49)
         DL = colonna (da 0 a 79)
         AL = numero da stampare
  Exit:  nessun valore in uscita
```

writeBin16: stampa sullo schermo un numero intero a 16 bit in base 2.

```
  Entry: DH = riga (da 0 a 49)
         DL = colonna (da 0 a 79)
         AX = numero da stampare
  Exit:  nessun valore in uscita
```

writeBin32: stampa sullo schermo un numero intero a 32 bit in base 2.

```
  Entry: DH = riga (da 0 a 49)
         DL = colonna (da 0 a 79)
         EAX = numero da stampare
  Exit:  nessun valore in uscita
```

readUdec8: legge dalla tastiera un numero intero senza segno a 8 bit in base 10.

```
  Entry: nessun parametro in ingresso
  Exit:  AL = numero letto
```

readUdec16: legge dalla tastiera un numero intero senza segno a 16 bit in base 10.

```
  Entry: nessun parametro in ingresso
  Exit:  AX = numero letto
```

readUdec32: legge dalla tastiera un numero intero senza segno a 32 bit in base 10.

```
  Entry: nessun parametro in ingresso
  Exit:  EAX = numero letto
```

readSdec8: legge dalla tastiera un numero intero con segno a 8 bit in base 10.

```
  Entry: nessun parametro in ingresso
  Exit:  AL = numero letto
```

readSdec16: legge dalla tastiera un numero intero con segno a 16 bit in base 10.

```
  Entry: nessun parametro in ingresso
  Exit:  AX = numero letto
```

readSdec32: legge dalla tastiera un numero intero con segno a 32 bit in base 10.

```
  Entry: nessun parametro in ingresso
  Exit:  EAX = numero letto
```

readHex8: legge dalla tastiera un numero intero a 8 bit in base 16.

```
  Entry: nessun parametro in ingresso
  Exit:  AL = numero letto
```

readHex16: legge dalla tastiera un numero intero a 16 bit in base 16.

```
  Entry: nessun parametro in ingresso
  Exit:  AX = numero letto
```

readHex32: legge dalla tastiera un numero intero a 32 bit in base 16.

```
  Entry: nessun parametro in ingresso
  Exit:  EAX = numero letto
```

readBin8: legge dalla tastiera un numero intero a 8 bit in base 2.

```
  Entry: nessun parametro in ingresso
  Exit:  AL = numero letto
```

```
readBin16: legge dalla tastiera un numero intero a 16 bit in base 2.
   Entry: nessun parametro in ingresso
   Exit:  AX = numero letto

readBin32: legge dalla tastiera un numero intero a 32 bit in base 2.
   Entry: nessun parametro in ingresso
   Exit:  EAX = numero letto

clearRegisters: azzera i registri EAX, EBX, ECX, EDX, ESI, EDI.
   Entry: nessun parametro in ingresso
   Exit:  nessun valore in uscita

showCPU: mostra in tempo reale lo stato della CPU.
   Entry: nessun parametro in ingresso
   Exit:  nessun valore in uscita
```

15.1.3 Ulteriori dettagli sulle librerie EXELIB e COMLIB

Le procedure presenti in **EXELIB** e **COMLIB**, sono molto facili da utilizzare; in ogni caso, vediamo alcuni chiarimenti relativi al loro funzionamento.

Le due procedure **set80x25** e **set80x50**, hanno lo scopo di selezionare la modalita' testo desiderata per lo schermo; **set80x25** attiva la modalita' video testo a **25** righe e **80** colonne, mentre **set80x50** attiva la modalita' video testo a **50** righe e **80** colonne.
Non appena si accende il computer, lo schermo viene inizializzato dal **BIOS** in modalita' testo standard; questa modalita' consiste nel gestire lo schermo attraverso una matrice formata da **25** righe (numerate da **0** a **24**) e **80** colonne (numerate da **0** a **79**), per un totale di **80x25=2000** celle.
Verso la fine degli anni **80**, la **IBM** ha introdotto la modalita' video grafica **VGA** standard, che consiste nel gestire lo schermo attraverso una matrice di **640x480** punti (pixel), ciascuno dei quali puo' assumere uno tra **16** colori differenti; trattandosi di uno standard, questa modalita' video viene sicuramente supportata da tutte le schede video **VGA** e superiori. La modalita' grafica **VGA**, rende disponibile una nuova modalita' testo formata da **50** righe e **80** colonne; in questo modo, e' possibile visualizzare il doppio delle informazioni rispetto alla modalita' testo a **25** righe. Le due procedure **set80x25** e **set80x50**, permettono appunto di selezionare la modalita' testo preferita; pur non essendo obbligatorio, si consiglia di utilizzare una di queste due procedure all'inizio di ogni programma che fa' uso della libreria **EXELIB** o **COMLIB**.

Nota importante
Nel caso in cui si stia utilizzando la **Dos Box** di **Windows XP**, e' importante che ogni programma che deve mostrare dei dati sullo schermo, inizi con le due procedure **set80x25** (o **set80x50**) e **clearScreen**; in caso contrario, puo' capitare che non si riesca a visualizzare nessun output sullo schermo!

La procedura **writeString**, permette di stampare una stringa sullo schermo, a partire dalla posizione indicata dai due registri **DH** (riga) e **DL** (colonna); seguendo la convenzione del linguaggio **C** (gia' illustrata in un precedente capitolo), la stringa da stampare deve terminare con un byte di valore zero (**NUL**). Per fare un esempio, la riga:

```
stringa_c db 'Linguaggio Assembly', 0,
```

definisce una stringa formata da **20** byte, con il ventesimo byte che vale zero (**NUL**); e' importante che non si faccia confusione tra il simbolo **NUL** (che ha codice ASCII **0**), e il simbolo **'0'** (che ha codice ASCII **48**).
Nel caso della libreria **EXELIB**, la procedura **writeString** riceve nei registri **ES:DI**, l'indirizzo

logico **Seg:Offset** della stringa da stampare; utilizzando la stringa dell'esempio precedente, prima di chiamare **writeString** possiamo scrivere:

```
mov      ax, seg stringa_c       ; ax = Seg(stringa_c)
mov      es, ax                  ; es = ax
mov      di, offset stringa_c    ; di = Offset(stringa_c)
```

In questo modo, la coppia **ES:DI** puntera' all'indirizzo logico **Seg:Offset** di **stringa_c**; come vedremo piu' avanti, nel caso della libreria **COMLIB**, viene richiesto solo il caricamento in **DI** della componente **Offset** dell'indirizzo della stringa da stampare.

Tutte le procedure del tipo **writeUdec**, **writeSdec**, **writeHex** e **writeBin**, permettono di stampare sullo schermo, numeri interi nelle principali ampiezze (**8**, **16** e **32** bit), e nelle principali basi (**10**, **16** e **2**); anche in questo caso, il programmatore, utilizzando i registri **DH** e **DL**, puo' stabilire in quale zona dello schermo verranno stampati i numeri.

Tutte le procedure del tipo **readUdec**, **readSdec**, **readHex** e **readBin**, permettono di inserire numeri interi dalla tastiera, utilizzando le tre ampiezze e le tre basi citate prima; l'input avviene nella sestultima riga dello schermo, e si consiglia pertanto di non sovrascrivere questa riga con l'output prodotto dalle procedure **writeString**, **writeHex**, etc.

Le procedure del tipo **readUdec** e **readSdec**, sono volutamente prive del controllo sui limiti del numero che viene inserito; in questo modo, e' possibile verificare quello che succede quando si inseriscono numeri fuori limite. La procedura **readUdec8**, ad esempio, dovrebbe ricevere in input un numero intero senza segno a **8** bit, compreso quindi tra **0** e **255**, ma in realta', accetta un numero intero senza segno, a tre cifre, compreso quindi tra **0** e **999**. Analogamente, la procedura **readSdec16**, dovrebbe ricevere in input un numero intero con segno a **16** bit, compreso quindi tra **-32768** e **+32767**, ma in realta', accetta un numero intero senza segno, a cinque cifre, compreso quindi tra **-99999** e **+99999**.
Per vedere allora quello che accade quando non rispettiamo i limiti, consideriamo il caso della procedura **readUdec8**; il numero inserito dal programmatore, viene restituito nel registro **AL** che conterra' quindi un valore binario, compreso tra **00000000b** e **11111111b**. Se proviamo ad inserire il numero **256** e a premere **[Invio]**, la procedura restituisce in **AL** il valore **0**; infatti, **256** (**100000000b**) e' un numero a **9** bit che per essere inserito negli **8** bit di **AL**, viene troncato a **00000000b** con perdita del bit piu' significativo.
Consideriamo ora il caso della procedura **readSdec16**; se proviamo ad inserire il numero **+32768** e a premere **[Invio]**, verra' restituito in **AX** il numero binario **1000000000000000b**, che rappresenta il valore **32768** per i numeri interi senza segno a **16** bit, e il valore **-32768** per i numeri interi con segno a **16** bit in complemento a due. Ricordiamo, infatti, che in complemento a due:

```
-32768 = 65536 - 32768 = 32768
```

Passando, infatti, il valore **1000000000000000b** alla procedura **writeUdec16**, verra' stampato il numero **32768**; passando invece il valore **1000000000000000b** alla procedura **writeSdec16**, verra' stampato proprio il numero **-32768**.

La procedura **showCPU** e' una delle piu' importanti della libreria, in quanto permette di visualizzare in tempo reale, lo stato della **CPU**, rappresentato dal contenuto di ogni suo registro; questa procedura, stampa le sue informazioni nelle ultime cinque righe dello schermo, e quindi, anche in questo caso, bisogna evitare di sovrascrivere quest'area di output.

15.2 Collegamento di una libreria ad un programma Assembly

Vediamo ora come si deve procedere per linkare ai nostri programmi, una libreria di procedure come **EXELIB** o **COMLIB**; tutti i dettagli su questo argomento, verranno illustrati in modo approfondito, in un apposito capitolo.

Supponiamo di scrivere un programma formato dai due file, **MAIN.ASM** (che contiene il programma principale) e **LIB1.ASM** (che contiene una libreria di procedure da utilizzare in **MAIN.ASM**); l'aspetto piu' importante da chiarire, riguarda il metodo che viene utilizzato per far comunicare tra loro **MAIN.ASM** e **LIB1.ASM**.
Come e' stato spiegato in un precedente capitolo, in **Assembly**, i nomi delle etichette, delle variabili, delle procedure, etc, hanno lo scopo di individuare la posizione (offset) di queste informazioni, calcolata rispetto al segmento di programma nel quale sono state definite le informazioni stesse; se, ad esempio, la procedura **writeString** inizia a partire dall'offset **0B00h** all'interno di un segmento di programma chiamato **CODESEGM**, allora, dal punto di vista della **CPU**, il nome **writeString** viene associato all'indirizzo logico **CODESEGM:0B00h**.
Se in **MAIN.ASM** e' presente un'istruzione che chiama la procedura **writeString** definita in **LIB1.ASM**, allora e' necessario indicare all'assembler, qual'e' l'indirizzo logico **Seg:Offset** della procedura stessa; infatti, chiamare una procedura, significa saltare all'indirizzo **Seg:Offset** da cui inizia in memoria la procedura stessa.

15.2.1 Le direttive PUBLIC e EXTRN

In sostanza, in fase di assemblaggio di un programma, ogni volta che l'assembler incontra una chiamata ad una procedura, deve essere in grado di convertire il nome di quella procedura, in una coppia **Seg:Offset** che in fase di esecuzione, verra' inserita in **CS:IP**; per fornire all'assembler tutte le informazioni di cui ha bisogno, vengono utilizzate le due direttive **PUBLIC** e **EXTRN**.

Applicando ad un nome la direttiva **PUBLIC**, si rende pubblico quel nome, cioe', lo si rende visibile anche all'esterno del file nel quale e' stato definito; se, ad esempio, in **LIB1.ASM** definiamo la procedura **writeString**, e poi scriviamo la dichiarazione:

```
PUBLIC writeString
```

allora la procedura **writeString** potra' essere chiamata, sia dall'interno del modulo **LIB1.ASM**, sia da un modulo esterno (come **MAIN.ASM**).
Tutti i nomi di procedure, etichette, variabili, etc, definiti in **LIB1.ASM** senza la direttiva **PUBLIC**, sono invisibili dall'esterno; in questo caso, si dice che questi nomi sono **privati**, nel senso che sono visibili (e quindi utilizzabili) solo all'interno di **LIB1.ASM**.

Se nel modulo **MAIN.ASM** vogliamo chiamare la procedura **writeString**, definita in **LIB1.ASM**, allora dobbiamo dire all'assembler che ci stiamo riferendo al nome di una procedura definita in un modulo esterno; per dare questa informazione all'assembler, dobbiamo servirci della direttiva **EXTRN**. Nel modulo **MAIN.ASM** dobbiamo quindi scrivere la dichiarazione:

```
EXTRN writeString: NEAR
```

Il termine **NEAR**, indica il tipo di indirizzamento necessario per accedere alla procedura che vogliamo chiamare; in relazione alla chiamata di una procedura che si trova in memoria all'indirizzo logico **Seg:Offset**, si possono presentare i seguenti due casi:

1) La procedura si trova nello stesso segmento di codice (ad esempio, **CODESEGM**) dal quale avviene la chiamata; in questo caso, la **CPU** deve modificare solamente **IP**, caricando in questo registro la componente **Offset** dell'indirizzo della procedura. Il registro **CS** non viene modificato in quanto, al momento della chiamata, contiene gia' la stessa componente **Seg** (**CODESEGM**)

dell'indirizzo della procedura; a questo punto, la **CPU** effettua un salto a **CS:IP**, chiamato **NEAR call** (chiamata vicina).

2) La procedura si trova in un segmento di codice (ad esempio, **CODESEGM2**) differente rispetto a quello (ad esempio, **CODESEGM**) dal quale avviene la chiamata; in questo caso, la **CPU** deve modificare, sia **CS**, sia **IP**. Nella coppia **CS:IP** viene caricato, ovviamente, l'indirizzo logico completo **Seg:Offset** della procedura; a questo punto, la **CPU** effettua un salto a **CS:IP**, chiamato **FAR call** (chiamata lontana).

Risulta evidente che gli indirizzi **NEAR**, essendo costituiti unicamente da un offset a **16** bit, vengono gestiti dalla **CPU** piu' velocemente rispetto agli indirizzi **FAR** (che comprendono, sia i **16** bit per la componente **Offset**, sia i **16** bit per la componente **Seg**); tutti questi concetti, sono stati illustrati in modo molto semplificato in quanto verranno spiegati in dettaglio in altri capitoli.

Per comodita', tutte le necessarie dichiarazioni **EXTRN** per le procedure esterne da utilizzare in un programma, possono essere inserite in appositi file, chiamati **include file**; convenzionalmente, per ogni include file relativo ad una libreria, viene utilizzato lo stesso nome della libreria, e l'estensione **INC**. Nel caso, ad esempio, della libreria **LIB1.ASM**, possiamo creare un apposito include file chiamato **LIB1.INC**; se il programma **MAIN.ASM** vuole servirsi delle procedure presenti in **LIB1.ASM**, deve avere al suo interno una direttiva del tipo:

```
INCLUDE LIB1.INC
```

Grazie a questa semplice direttiva, tutte le dichiarazioni **EXTRN** presenti in **LIB1.INC**, vengono incorporate nel modulo **MAIN.ASM**; piu' avanti vedremo ulteriori dettagli su questo argomento.

15.2.2 Assembling & linking di un programma diviso in due o piu' moduli

Una volta che e' stato chiarito il metodo che permette a **MAIN.ASM** e **LIB1.ASM** di comunicare tra loro, possiamo passare alle fasi di assembling e linking, che ci permettono di ottenere il programma finale in versione eseguibile; prima di tutto, bisogna procedere all'assemblaggio separato dei due file.

Con il **TASM**, dobbiamo impartire i comandi:

```
..\bin\tasm main.asm
```

e:

```
..\bin\tasm lib1.asm
```

oppure:

```
..\bin\tasm main.asm + lib1.asm
```

Con il **MASM**, dobbiamo impartire i comandi:

```
..\bin\ml /c main.asm
```

e:

```
..\bin\ml /c lib1.asm
```

oppure:

```
..\bin\ml /c main.asm + lib1.asm
```

Se non si sono verificati errori, abbiamo ottenuto i due file in formato oggetto **MAIN.OBJ** e **LIB1.OBJ**, e possiamo quindi passare alla fase di linking; con il **TASM**, dobbiamo impartire il comando:

```
..\bin\tlink main.obj + lib1.obj
```

Con il **MASM**, dobbiamo impartire il comando:

```
..\bin\link main.obj + lib1.obj
```

Il comportamento predefinito del linker, e' quello di assegnare all'eseguibile, il nome del file piu' a sinistra tra quelli che il programmatore ha specificato dalla linea di comando; nel nostro caso, si

ottiene un eseguibile chiamato **MAIN.EXE**.
Se avessimo scritto, invece:
```
..\bin\tlink lib1.obj + main.obj
```
il linker avrebbe assegnato all'eseguibile il nome **LIB1.EXE**; piu' avanti vedremo ulteriori dettagli sulle fasi di assembling e di linking di un programma suddiviso in due o piu' moduli.

15.3 L'istruzione CALL

In **Assembly**, il metodo predefinito per la chiamata di una procedura, consiste nell'uso dell'istruzione **CALL**, che verra' illustrata in dettaglio in un apposito capitolo; per il momento, ci limitiamo ad una semplice descrizione di questa istruzione, in modo da poterla utilizzare negli esempi presentati piu' avanti.

Nella sua forma piu' semplice, la chiamata di una procedura viene definita come **direct call within segment** (chiamata diretta all'interno del segmento); come e' stato spiegato in precedenza, questa situazione si verifica quando la procedura si trova nello stesso segmento dal quale avviene la chiamata. In un caso del genere, la **CPU** ha bisogno di conoscere solo la componente **Offset** dell'indirizzo della procedura, in modo da porre **IP=Offset**; il registro **CS** non deve essere modificato, in quanto contiene gia' una componente **Seg** che e' la stessa dell'indirizzo della procedura.
Il codice macchina di questa forma dell'istruzione **CALL**, e' composto dall'**Opcode 11101000b** (**E8h**), seguito da un valore a **16** bit; la **CPU**, somma questo valore all'offset dell'istruzione successiva alla **CALL**, e ottiene la componente **Offset** dell'indirizzo della procedura da chiamare.
Per chiarire questo concetto, vediamo un esempio pratico; supponiamo di avere le seguenti due istruzioni:

```
call     writeString
mov      ax, bx
```

e supponiamo che la prima istruzione si trovi all'offset **00BBh** di un segmento di programma chiamato **CODESEGM**. Il codice macchina dell'istruzione **CALL** e' formato dall'**Opcode E8h** (**1** byte), seguito da un valore a **16** bit (**2** byte), per un totale di **3** byte; di conseguenza, l'istruzione successiva alla **CALL** si trova all'offset:
```
00BBh + 0003h = 00BEh
```
Supponiamo ora che la procedura **writeString** parta dall'offset **0AC2h** nello stesso blocco **CODESEGM**; di conseguenza, l'assembler tradurra' le due istruzioni precedenti, nel codice macchina:

```
00BBh    E8h 0A04h      ; call   writeString
00BEh    8Bh C3h        ; mov    ax, bx
```

Quando la **CPU** arriva all'offset **00BBh** del blocco **CODESEGM**, incontra l'istruzione contenente la **NEAR call**, e calcola:
```
0A04h + 00BEh = 0AC2h
```
che e' proprio l'offset da cui inizia **writeString**; la **CPU** quindi carica **0AC2h** in **IP**, e salta a **CS:IP**, cioe' a **CODESEGM:0AC2h** (**NEAR call**).

Cosa succede quando **writeString** termina il suo lavoro?
E' chiaro che se non si prendono delle precauzioni, la **CPU** non e' in grado di sapere da quale punto riprendera' l'esecuzione del programma; osservando il codice macchina illustrato nell'esempio precedente, risulta evidente che una volta che **writeString** ha terminato il suo lavoro, l'esecuzione del programma dovra' riprendere dall'istruzione successiva alla **CALL**, e quindi dall'offset **00BEh**.

In base a queste considerazioni, la **CPU**, quando incontra l'istruzione per la **NEAR call** di **writeString**, esegue le seguenti operazioni:

1) sottrae **2** a **SP** per fare spazio a una nuova **WORD** nello stack;
2) salva all'indirizzo **SS:SP**, l'offset (**00BEh**) dell'istruzione successiva alla **CALL**;
3) carica in **IP** l'offset dell'indirizzo di **writeString**, e salta a **CS:IP**.

Come si vedra' in un apposito capitolo, una procedura termina con l'istruzione **RET** (return from procedure); quando la **CPU** incontra una istruzione **RET** (che nel nostro caso e' un **NEAR return** da **writeString**), esegue in sequenza le seguenti operazioni:

1) estrae un valore a **16** bit dall'indirizzo **SS:SP** dello stack, e lo carica in **IP**;
2) incrementa **SP** di **2**, in modo da recuperare lo spazio appena liberato nello stack;
3) salta all'indirizzo **CS:IP**.

Se il programmatore non ha commesso errori nella gestione dello stack (all'interno di **writeString**), il valore a **16** bit che la **CPU** estrae dall'indirizzo **SS:SP** rappresenta proprio l'offset **00BEh** salvato in precedenza dalla **CPU** stessa; di conseguenza, l'indirizzo della prossima istruzione da eseguire, contenuto in **CS:IP**, sara' proprio quello dell'istruzione successiva alla **CALL**.

Analizziamo ora un'altra forma della chiamata di una procedura, che viene definita come **direct call intersegment** (chiamata diretta intersegmento); come e' stato spiegato in precedenza, questa situazione si verifica quando la procedura si trova in un segmento diverso da quello nel quale avviene la chiamata. In un caso del genere, la **CPU** ha bisogno di conoscere l'indirizzo completo **Seg:Offset** della procedura, in modo da porre **CS=Seg** e **IP=Offset**; a causa del salto da un segmento ad un altro, e' necessario quindi modificare anche **CS**.
Il codice macchina di questa forma dell'istruzione **CALL**, e' costituito dall'**Opcode 10011010b** (**9Ah**), seguito da un valore a **32** bit; questo valore a **32** bit, rappresenta l'indirizzo completo **Seg:Offset** della procedura da chiamare. Per chiarire questo concetto, vediamo un esempio pratico; supponiamo di avere le seguenti due istruzioni:

```
call     writeString
mov      ax, bx
```

e supponiamo che la prima istruzione si trovi all'offset **00BBh** di un segmento di programma chiamato **CODESEGM**. Il codice macchina dell'istruzione **CALL** e' formato dall'**Opcode 9Ah** (**1** byte), seguito da un valore a **32** bit (**4** byte), per un totale di **5** byte; di conseguenza, l'istruzione successiva alla **CALL** si trova all'offset:

```
00BBh + 0005h = 00C0h
```

Supponiamo ora che la procedura **writeString** parta dall'offset **0AC2h** di un blocco codice chiamato **CODESEGM2**; di conseguenza, l'assembler tradurra' le due istruzioni precedenti nel codice macchina:

```
00BBh    9Ah 00000000h     ; call   writeString
00C0h    8Bh C3h           ; mov    ax, bx
```

Notiamo che se l'assembler non e' in grado di determinare in anticipo l'indirizzo di una procedura (definita, ad esempio, in un modulo esterno), delega questo compito al linker.
Quando la **CPU** arriva all'offset **00BBh** del blocco **CODESEGM**, incontra l'istruzione contenente la **FAR call**, ed esegue le seguenti operazioni:

1) sottrae **4** a **SP** per fare spazio a una nuova **DWORD** nello stack;
2) salva all'indirizzo **SS:SP**, l'indirizzo completo **CODESEGM:00C0h** dell'istruzione successiva alla **CALL**;
3) carica in **CS:IP** l'indirizzo completo (**CODESEGM2:0AC2h**) di **writeString**, e salta a **CS:IP**.

Al termine di **writeString**, e' presente la solita istruzione **RET**, che in questo caso e' un **FAR return** da **writeString**; quando la **CPU** incontra l'istruzione **RET**, esegue in sequenza le seguenti operazioni:

1) estrae un valore a **32** bit dall'indirizzo **SS:SP** dello stack, e lo carica in **CS:IP**;
2) incrementa **SP** di **4**, in modo da recuperare lo spazio appena liberato nello stack;
3) salta all'indirizzo **CS:IP**.

Se il programmatore non ha commesso errori nella gestione dello stack (all'interno di **writeString**), il valore a **32** bit che la **CPU** estrae dall'indirizzo **SS:SP**, rappresenta proprio l'indirizzo **CODESEGM:00C0h** salvato in precedenza dalla **CPU** stessa; di conseguenza, l'indirizzo della prossima istruzione da eseguire, contenuto in **CS:IP**, sara' proprio quello dell'istruzione successiva alla **CALL**.

E' importantissimo ribadire ancora una volta che in **Assembly**, il programmatore ha il controllo diretto su molti delicati aspetti legati al programma in esecuzione; in particolare, il programmatore ha il dovere di gestire correttamente lo stack, altrimenti la **CPU** non e' in grado di eseguire con successo le fasi descritte negli esempi precedenti. Se, ad esempio, all'interno di **writeString** il programmatore salva con **PUSH** il contenuto di un registro, e poi dimentica di ripristinare lo stack con **POP**, quello che accade e' che quando la **CPU** incontra l'istruzione **RET**, invece di estrarre dallo stack l'indirizzo della prossima istruzione da eseguire, estrae quella che, in gergo, viene definita "spazzatura"; la **CPU** carica questa spazzatura in **CS:IP**, e compie un "salto nel buio"! Generalmente, questa situazione manda in crash il programma in esecuzione; per il momento comunque, non bisogna preoccuparsi di questi aspetti, perche' verranno spiegati in dettaglio in altri capitoli.

15.4 Programma di prova EXETEST2.ASM per la libreria EXELIB

Dopo aver esposto tutti i necessari concetti teorici, possiamo passare ad un esempio pratico, rappresentato da un programma che ci permettera' di testare le procedure contenute nella libreria **EXELIB.OBJ**; a tale proposito, ci serviremo del programma **EXETEST2.ASM**, illustrato dalla Figura 1.

Figura 1 - File EXETEST2.ASM

```
;--------------------------------------------------;
; File exetest2.asm                                ;
; Verifica a run-time di un programma EXE          ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

INCLUDE     EXELIB.INC              ; inclusione libreria di I/O
.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                  ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;----- inizio definizione variabili statiche ------

strTitle    db    'PROGRAMMA DI PROVA EXETEST2.ASM '
            db    'PER LA LIBRERIA EXELIB', 0
strExit     db    'Premere un tasto per terminare', 0

;------- fine definizione variabili statiche ------

DATASEGM    ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; associa CODESEGM a CS

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; associa DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   call     set80x50                ; modo video testo 80x50
   call     hideCursor              ; nasconde il cursore
   call     clearScreen             ; pulisce lo schermo

   call     clearRegisters          ; azzera i registri generali
   call     showCPU                 ; mostra lo stato della CPU

   mov      dx, 000eh               ; riga 00h, colonna 0Eh
   mov      ax, seg strTitle        ; AX = Seg(strTitle)
```

```
mov     es, ax                  ; ES = AX
mov     di, offset strTitle     ; ES:DI punta a strTitle
call    writeString             ; stampa la stringa

call    readUdec8               ; legge un int. senza segno a 8 bit
mov     dx, 0300h               ; riga 03h, colonna 00h
call    writeUdec8              ; stampa il numero letto (AL)
call    showCPU                 ; mostra lo stato della CPU

call    readUdec16              ; legge un int. senza segno a 16 bit
mov     dx, 0400h               ; riga 04h, colonna 00h
call    writeUdec16             ; stampa il numero letto (AX)
call    showCPU                 ; mostra lo stato della CPU

call    readUdec32              ; legge un int. senza segno a 32 bit
mov     dx, 0500h               ; riga 05h, colonna 00h
call    writeUdec32             ; stampa il numero letto (EAX)
call    showCPU                 ; mostra lo stato della CPU

call    clearRegisters          ; azzera i registri generali

call    readSdec8               ; legge un int. con segno a 8 bit
mov     dx, 0700h               ; riga 07h, colonna 00h
call    writeSdec8              ; stampa il numero letto (AL)
call    showCPU                 ; mostra lo stato della CPU

call    readSdec16              ; legge un int. con segno a 16 bit
mov     dx, 0800h               ; riga 08h, colonna 00h
call    writeSdec16             ; stampa il numero letto (AX)
call    showCPU                 ; mostra lo stato della CPU

call    readSdec32              ; legge un int. con segno a 32 bit
mov     dx, 0900h               ; riga 09h, colonna 00h
call    writeSdec32             ; stampa il numero letto (EAX)
call    showCPU                 ; mostra lo stato della CPU

call    clearRegisters          ; azzera i registri generali

call    readHex8                ; legge un intero hex. a 8 bit
mov     dx, 0b00h               ; riga 0Bh, colonna 00h
call    writeHex8               ; stampa il numero letto (AL)
call    showCPU                 ; mostra lo stato della CPU

call    readHex16               ; legge un intero hex. a 16 bit
mov     dx, 0c00h               ; riga 0Ch, colonna 00h
call    writeHex16              ; stampa il numero letto (AX)
call    showCPU                 ; mostra lo stato della CPU

call    readHex32               ; legge un intero hex. a 32 bit
mov     dx, 0d00h               ; riga 0Dh, colonna 00h
call    writeHex32              ; stampa il numero letto (EAX)
call    showCPU                 ; mostra lo stato della CPU

call    clearRegisters          ; azzera i registri generali

call    readBin8                ; legge un intero bin. a 8 bit
mov     dx, 0f00h               ; riga 0Fh, colonna 00h
call    writeBin8               ; stampa il numero letto (AL)
call    showCPU                 ; mostra lo stato della CPU

call    readBin16               ; legge un intero bin. a 16 bit
mov     dx, 1000h               ; riga 10h, colonna 00h
call    writeBin16              ; stampa il numero letto (AX)
```

```
   call     showCPU                  ; mostra lo stato della CPU

   call     readBin32                ; legge un intero bin. a 32 bit
   mov      dx, 1100h                ; riga 11h, colonna 00h
   call     writeBin32               ; stampa il numero letto (EAX)
   call     showCPU                  ; mostra lo stato della CPU

   mov      dx, 1400h                ; riga 14h, colonna 00h
   mov      ax, seg strExit          ; AX = Seg(strExit)
   mov      es, ax                   ; ES = AX
   mov      di, offset strExit       ; ES:DI punta a strExit
   call     writeString              ; stampa la stringa

   call     waitChar                 ; attende la pressione di un tasto
   call     clearScreen              ; pulisce lo schermo
   call     showCursor               ; ripristina il cursore

;-------- fine blocco principale istruzioni -------

   mov      ah, 4ch                  ; servizio Terminate Program
   mov      al, 00h                  ; exit code = 0
   int      21h                      ; chiama i servizi DOS

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

Analizzando il listato di **EXETEST2.ASM**, notiamo che la prima riga contiene la direttiva:
`INCLUDE EXELIB.INC`
che permette di includere, comodamente, tutte le necessarie direttive **EXTRN**, relative alle procedure definite nel modulo **EXELIB.OBJ**. Aprendo con un editor il file **EXELIB.INC**, possiamo notare che tutte le procedure vengono dichiarate di tipo **FAR**; infatti, queste procedure vengono definite, nel modulo **EXELIB.OBJ**, all'interno di un apposito segmento di codice, che e' diverso dal segmento **CODESEGM** presente nel modulo **EXETEST2.ASM**.
Siccome non conosciamo il nome del segmento di codice, interno a **EXELIB.OBJ**, nel quale sono state definite le varie procedure esterne, e' importante che le relative direttive **EXTRN**, vengano poste in un'area del modulo **EXETEST2.ASM**, che deve trovarsi al di fuori di qualsiasi segmento di programma; come si nota in Figura 1, il punto piu' adatto per l'inserimento di queste direttive (poste nel file **EXELIB.INC**), e' rappresentato dalla parte iniziale del modulo che contiene il programma principale.

Nota importante.
Bisogna prestare particolare attenzione al fatto che, nel caso del **MASM**, le direttive **EXTRN** relative a procedure esterne, devono essere inserite prima di una eventuale direttiva **.386**; se il **MASM** incontra per prima la direttiva **.386**, crede di trovarsi davanti ad un programma scritto per la modalita' protetta. In tal caso, il **MASM**, in presenza di una chiamata ad una procedura **FAR**, genera un codice macchina del tipo:
66h 9Ah 00000000h
Questo codice macchina, inizia con il valore **66h** che, come sappiamo, indica la presenza di operandi a **32** bit; quando la **CPU** incontra questo codice macchina, effettua un salto ad un indirizzo **FAR**, tipico della modalita' protetta, formato da una componente **Seg** a **16** bit, e da una componente **Offset** a **32** bit!
In modalita' reale, questo tipo di salto provoca un sicuro crash del programma; proprio per questo motivo, in Figura 1 la direttiva **INCLUDE** e' stata posta prima della direttiva **.386**.

Nel blocco **DATASEGM** di Figura 1, vengono definite due stringhe **C**, chiamate **strTitle** e **strExit**; l'unico aspetto degno di nota, e' rappresentato dal metodo che viene utilizzato per definire stringhe molto lunghe. Anziche' definire una stringa su una sola gigantesca linea, si puo' fare ricorso al metodo che e' stato seguito in Figura 1 per **strTitle**; come sappiamo, l'assembler provvede a disporre in memoria, in modo consecutivo e contiguo, tutti i byte che formano **strTitle**.

Nel blocco **CODESEGM** di Figura 1, vengono chiamate quasi tutte le procedure definite in **EXELIB.OBJ**; in questo modo, e' possibile analizzare il loro funzionamento.
Come possiamo notare, vengono inizialmente chiamate, **set80x50** (che seleziona la modalita' video testo), **hideCursor** (che nasconde il cursore lampeggiante) e **clearScreen** (che cancella lo schermo); subito dopo, viene chiamata la procedura **clearRegisters**, che provvede ad azzerare i registri **EAX**, **EBX**, **ECX**, **EDX**, **ESI** e **EDI**.

Tutte le procedure definite in **EXELIB.OBJ**, ad eccezione di **showCPU**, preservano i registri (a **8**, **16** e **32** bit) che utilizzano; la procedura **clearRegisters** non preserva, ovviamente, il contenuto dei **6** registri da azzerare. Questo aspetto diventa molto importante, nel momento in cui non vogliamo che la chiamata di una procedura, alteri il contenuto di determinati registri che stiamo utilizzando nel programma principale; eventualmente, e' opportuno che vengano prese le necessarie precauzioni. Supponiamo, ad esempio, di voler chiamare una procedura **Proc1**, che altera il registro **AX** senza preservarne il vecchio contenuto; se, nel programma principale, non vogliamo perdere il vecchio contenuto di **AX**, dobbiamo scrivere istruzioni del tipo:

```
push     ax       ; preserva il contenuto di AX
call     Proc1    ; chiama Proc1
pop      ax       ; ripristina il contenuto di AX
```

Subito dopo **clearRegisters**, viene chiamata la procedura **showCPU**, che visualizza, in tempo reale, il contenuto di tutti i registri della **CPU**; grazie a **showCPU**, possiamo analizzare, in particolare, il contenuto iniziale di **CS**, **DS**, **ES**, **SS**, **SP** e **IP**, in modo da conoscere tutti i dettagli relativi all'area della memoria, nella quale e' stato caricato il nostro programma per la fase di esecuzione.

Proseguendo nell'analisi del listato di Figura 1, fissiamo la nostra attenzione sulla chiamata della procedura **writeString** per la visualizzazione della stringa **strTitle**; nel caso della libreria **EXELIB.OBJ**, la procedura **writeString** si trova in un modulo esterno, e non ha la piu' pallida idea di quali siano le caratteristiche dei segmenti di codice, dati e stack, definiti nel modulo **EXETEST2.ASM**. Proprio per questo motivo, **writeString** ha bisogno di conoscere l'indirizzo

completo **Seg:Offset** della stringa da visualizzare; questa informazione, come viene mostrato dalla Figura 2, deve essere passata attraverso la coppia **ES:DI**.

Figura 2 - Passaggio degli argomenti a writeString

```
mov     dx, 000eh              ; riga 00h, colonna 0Eh
mov     ax, seg strTitle       ; AX = Seg(strTitle)
mov     es, ax                 ; ES = AX
mov     di, offset strTitle    ; ES:DI punta a strTitle
call    writeString            ; stampa la stringa
```

Osserviamo il metodo che viene seguito per caricare, in un colpo solo, il valore **00h** in **DH**, e il valore **0Eh** in **DL**; in base alle proprieta' dei numeri esadecimali, espressi nel sistema posizionale, le due cifre piu' significative (**00h**) di **000Eh**, si posizionano nei **16** bit piu' significativi (**DH**) di **DX**, mentre le due cifre meno significative (**0Eh**) di **000Eh**, si posizionano nei **16** bit meno significativi (**DL**) di **DX**.
La Figura 2 ci permette anche di illustrare in pratica, la differenza concettuale che esiste tra i **parametri in ingresso** e gli **argomenti in ingresso**; i **parametri** che **writeString** richiede in ingresso, sono **DH**, **DL**, **ES** e **DI**, mentre gli **argomenti** che il programmatore passa a **writeString** in ingresso, sono, rispettivamente, **00h**, **0Eh**, **seg strTitle** e **offset strTitle**.

Come e' stato gia' detto, tutte le procedure del tipo **readUdec**, **readSdec**, **readHex** e **readBin**, utilizzano la sestultima riga per mostrare i numeri che l'utente inserisce dalla tastiera; inoltre, la riga precedente a quella di input, mostra una stringa che fornisce informazioni sul tipo e sui limiti inferiore e superiore dei numeri da inserire.

15.4.1 Assembling & Linking del programma EXETEST2

Dopo aver chiarito questi aspetti, possiamo passare alle fasi di assembling e di linking del nostro programma.
Come e' stato detto in precedenza, la fase di assembling coinvolge tutti i moduli che contengono il codice sorgente di un programma distribuito su due o piu' file; nel nostro caso, il modulo **EXELIB.OBJ** e' gia' in formato **object code**, per cui, dobbiamo procedere con l'assemblaggio del solo modulo **EXETEST2.ASM**.

Con il **TASM**, il comando da impartire e':
```
..\bin\tasm /zn /l exetest2.asm
```
Se lo preferiamo, possiamo anche richiedere al **TASM** un assemblaggio **case-sensitive**, in modo da rispettare la distinzione tra lettere maiuscole e lettere minuscole negli identificatori; in tal caso, e' necessario servirsi dell'opzione **/ml**, impartendo quindi il comando:
```
..\bin\tasm /ml /zn /l exetest2.asm
```

Con il **MASM**, il comando da impartire e':
```
..\bin\ml /c /Fl exetest2.asm
```
Se vogliamo un assemblaggio **case-sensitive**, dobbiamo servirci dell'opzione **/Cp**, impartendo quindi il comando:
```
..\bin\ml /c /Cp /Fl exetest2.asm
```

Premendo ora il tasto **[Invio]**, vengono generati i due file **EXETEST2.OBJ** e **EXETEST2.LST**; apriamo subito con un editor, il file **EXETEST2.LST** per poter esaminare alcuni aspetti interessanti.
Nel **listing file** notiamo subito che tutte le direttive **EXTRN** presenti in **EXELIB.INC**, sono state incorporate in **EXETEST2.LST**; inoltre, le stesse direttive **EXTRN** sono state sistemate al di fuori

di qualsiasi segmento di programma presente in **EXETEST2.LST**. Cio' permette all'assembler di generare l'opportuno codice macchina relativo alla chiamata delle varie procedure; osserviamo infatti che, ad esempio, in relazione alla chiamata di **set80x50**, viene generato il codice macchina:
`9Ah 00000000h` (**FAR call**).
Inoltre, nella **symbol table**, la procedura **set80x50** viene classificata dall'assembler come:
```
set80x50 Far ----:---- Extern
```
In questo caso, l'assembler sta delegando al linker, il compito di calcolare l'indirizzo logico **Seg:Offset** della procedura esterna **set80x50** di tipo **Far**.

A titolo di curiosita', vediamo quello che succede con **TASM**, se inseriamo la direttiva:
```
INCLUDE EXELIB.INC
```
all'interno del blocco **CODESEGM** di **EXETEST2.ASM**; in questo caso, esaminando il file **EXETEST2.LST**, ci accorgiamo che nella **symbol table**, l'assembler ha inserito informazioni del tipo:
```
set80x50 Far CODESEGM:---- Extern
```
In sostanza, **TASM** sta dicendo al linker, che e' necessario calcolare la sola componente **Offset** dell'indirizzo logico della procedura esterna **set80x50**, in quanto la componente **Seg** e' sicuramente **CODESEGM**; l'assembler quindi, e' stato tratto in inganno dal fatto che abbiamo inserito la direttiva **INCLUDE** nel punto sbagliato.
Tutti questi aspetti, molto delicati, verranno analizzati in dettaglio in un apposito capitolo.

Passiamo ora alla fase di linking del nostro programma; con il **TASM**, il comando da impartire e':
```
..\bin\tlink exetest2.obj + exelib.obj
```

Con il **MASM**, il comando da impartire e':
```
..\bin\link /map exetest2.obj + exelib.obj
```

Premendo il tasto **[Invio]**, parte la fase di linking che, in assenza di errori, porta alla generazione del file in formato eseguibile **EXETEST2.EXE**, e del map file **EXETEST2.MAP**; analizziamo il lavoro svolto dal linker per il collegamento dei due moduli **EXETEST2.OBJ** e **EXELIB.OBJ**.

Prima di tutto, bisogna premettere che il modulo **EXELIB.OBJ**, contiene un segmento dati del tipo:
```
LIBIODATA SEGMENT PARA PRIVATE USE16 'DATA'
```
e un segmento di codice del tipo:
```
LIBIOCODE SEGMENT PARA PUBLIC USE16 'CODE'
```

1) Il linker inizia ad esaminare il modulo **EXETEST2.OBJ**, dove trova il primo segmento **DATASEGM**.
2) Il linker cerca un eventuale altro segmento **DATASEGM** nel modulo **EXELIB.OBJ**, e non trovandolo, crea il primo blocco di programma, rappresentato dallo stesso **DATASEGM** con allineamento **PARA**.
3) Il linker cerca altri segmenti di classe **'DATA'**, e trova il segmento **LIBIODATA** nel modulo **EXELIB.OBJ**.
4) Non esistendo altri segmenti **LIBIODATA**, il linker crea il secondo blocco di programma, rappresentato dallo stesso **LIBIODATA** con allineamento **PARA**.
5) Non esistendo altri segmenti di classe **'DATA'**, il linker torna al modulo **EXETEST2.OBJ**, dove trova il segmento **CODESEGM**.
6) Il linker cerca un eventuale altro segmento **CODESEGM** nel modulo **EXELIB.OBJ**, e non trovandolo, crea il terzo blocco di programma, rappresentato dallo stesso **CODESEGM** con allineamento **PARA**.
7) Il linker cerca altri segmenti di classe **'CODE'**, e trova il segmento **LIBIOCODE** nel modulo

EXELIB.OBJ.
8) Non esistendo altri segmenti **LIBIOCODE**, il linker crea il quarto blocco di programma, rappresentato dallo stesso **LIBIOCODE** con allineamento **PARA**.
9) Non esistendo altri segmenti di classe **'CODE'**, il linker torna al modulo **EXETEST2.OBJ**, dove trova il segmento **STACKSEGM**.
10) Il linker cerca un eventuale altro segmento **STACKSEGM** nel modulo **EXELIB.OBJ**, e non trovandolo, crea il quinto blocco di programma, rappresentato dallo stesso **STACKSEGM** con allineamento **PARA**.
11) Il linker non trova altri segmenti, per cui la fase di linking e' terminata.

Le considerazioni appena esposte, vengono confermate dal **map file** di Figura 3.

Figura 3 - EXETEST2.MAP

```
Start   Stop    Length  Name         Class

00000h  00055h  00056h  DATASEGM     DATA
00060h  0041Bh  003BCh  LIBIODATA    DATA
00420h  00559h  0013Ah  CODESEGM     CODE
00560h  0113Ah  00BDBh  LIBIOCODE    CODE
01140h  0153Fh  00400h  STACKSEGM    STACK

Program entry point at 0042h:0000h
```

Il linker ha calcolato per l'**entry point**, l'indirizzo logico **0042h:0000h**; osserviamo, infatti, che il blocco **CODESEGM** parte dall'indirizzo fisico **00420h** (allineato al paragrafo), a cui corrisponde l'indirizzo logico normalizzato **0042h:0000h**. Nel listato di Figura 1, notiamo che l'etichetta **start** si trova all'offset **0000h** di **CODESEGM**; quest'offset non viene rilocato dal linker, per cui, l'indirizzo logico dell'**entry point** e' proprio **0042h:0000h**.
In Figura 3, possiamo anche notare che grazie all'attributo **Class**, il linker ha potuto stabilire il criterio di ordinamento per i vari blocchi di memoria; infatti, il nostro programma contiene, prima i blocchi di classe **'DATA'**, poi i blocchi di classe **'CODE'**, e infine il blocco di classe **'STACK'**.

Cosa succede se effettuiamo il linking con il comando:
`..\bin\tlink exelib.obj + exetest2.obj`?
In tal caso, il linker inizia ad esaminare per primo il modulo **EXELIB.OBJ**; il primo segmento che viene incontrato e' **LIBIODATA**. Alla fine, si ottiene un eseguibile che viene chiamato **EXELIB.EXE**, e un **map file** che viene chiamato **EXELIB.MAP**; la struttura interna di **EXELIB.EXE**, viene illustrata proprio dal **map file** di Figura 4.

Figura 4 - EXELIB.MAP

```
Start   Stop    Length  Name         Class

00000h  003BBh  003BCh  LIBIODATA    DATA
003C0h  00415h  00056h  DATASEGM     DATA
00420h  00FFAh  00BDBh  LIBIOCODE    CODE
01000h  01139h  0013Ah  CODESEGM     CODE
01140h  0153Fh  00400h  STACKSEGM    STACK

Program entry point at 0100h:0000h
```

Come si puo' notare, questa volta il blocco **CODESEGM** viene fatto partire dall'indirizzo fisico **01000h**, a cui corrisponde l'indirizzo logico normalizzato **0100h:0000h**; di conseguenza, l'**entry point** calcolato dal linker, si viene a trovare all'indirizzo logico **0100h:0000h**.

15.4.2 Esecuzione del programma EXETEST2.EXE

Passiamo ora alla fase di esecuzione del file **EXETEST2.EXE**; posizionandoci nella cartella **ASMBASE**, dal **prompt** del **DOS** impartiamo il comando:

```
exetest2
```

e premiamo il tasto **[Invio]**.
Chi lavora in ambiente **Windows**, puo' aprire diverse **Dos-Box**, attraverso le quali si puo' avere un controllo totale della situazione; ad esempio, in una **Dos-Box** si puo' aprire il file **EXETEST2.LST**, in un'altra **Dos-Box** si puo' aprire il file **EXETEST2.MAP**, in un'altra **Dos-Box** si puo' eseguire il file **EXETEST2.EXE** e cosi' via.

Appena inizia l'esecuzione di **EXETEST2.EXE**, vedremo comparire in fondo allo schermo, una serie di informazioni come quelle mostrate in Figura 5; queste informazioni, si riferiscono alla richiesta di input che arriva dalla procedura **readUdec8**, e allo stato della **CPU** mostrato dalla procedura **showCPU**.

Figura 5 - EXETEST2.EXE in esecuzione
Inserire un numero in base 10 tra 0 e 255:
CPU
EAX=00000000H EBX=00000000H ECX=00000000H EDX=00000000H ESI=00000000H EDI=00000000H ESP=00000400H EBP=00000912H CS=0CF7H DS=0CB5H ES=0CA5H FS=0CB5H GS=0000H SS=0DC9H IP=0019H CF = 0 PF = 1 AF = 0 ZF = 1 SF = 0 TF = 0 IF = 1 DF = 0 OF = 0

La procedura **showCPU**, chiamata proprio all'inizio del nostro programma, mostra il contenuto iniziale dei registri della **CPU**; analizziamo, in particolare, l'importante contenuto iniziale dei registri **CS**, **DS**, **ES**, **SS**, **SP** e **IP** (si tenga presente che le informazioni contenute nei registri di segmento, possono variare da computer a computer).
Il registro **ES**, non ha subito ancora nessuna modifica, per cui, contiene di sicuro la componente **Seg** dell'indirizzo logico iniziale del **program segment** assegnato a **EXETEST2.EXE**; possiamo dire allora che il nostro programma, parte in memoria dall'indirizzo logico **0CA5h:0000h**, a cui corrisponde l'indirizzo fisico **0CA50h**.

I primi **256** byte (**0100h** byte) del **program segment**, sono occupati dal **PSP**; in base al **map file** di Figura 3, il primo blocco del nostro programma e' **DATASEGM**, che quindi parte dall'indirizzo fisico:

```
0CA50h + 0100h = 0CB50h
```

All'indirizzo fisico **0CB50h**, corrisponde l'indirizzo logico normalizzato **0CB5h:0000h**; la componente **0CB5h** di questo indirizzo, viene assegnata a **DATASEGM**. Prima della chiamata di **showCPU**, il valore **DATASEGM** e' stato assegnato a **DS** (vedere il listato di Figura 1); infatti, in Figura 5 vediamo che **DS=0CB5h**.

In base al **map file** di Figura 3, il blocco **CODESEGM** si trova a **00420h** byte di distanza dall'inizio di **DATASEGM**; possiamo dire allora che **CODESEGM** parte in memoria dall'indirizzo fisico:

```
0CB50h + 00420h = 0CF70h
```

All'indirizzo fisico **0CF70h**, corrisponde l'indirizzo logico normalizzato **0CF7h:0000h**, che coincide anche con l'indirizzo logico dell'**entry point**; la componente **0CF7h** di questo indirizzo, viene assegnata a **CODESEGM**. Non appena inizia l'esecuzione del nostro programma, il valore **CODESEGM** viene utilizzato quindi per inizializzare **CS**; infatti, in Figura 5 vediamo che **CS=0CF7h**.
Il valore (**0019h**) di **IP** visibile in Figura 5, si riferisce al contenuto dell'**instruction pointer** al momento della prima chiamata di **showCPU**; infatti, analizzando il file **EXETEST2.LST**, si scopre

che la prima chiamata di **showCPU**, si trova proprio all'indirizzo logico **CODESEGM:0019h**.

In base al **map file** di Figura 3, il blocco **STACKSEGM** si trova a **01140h** byte di distanza dall'inizio di **DATASEGM**; possiamo dire allora che **STACKSEGM** parte in memoria dall'indirizzo fisico:
`0CB50h + 01140h = 0DC90h`
All'indirizzo fisico **0DC90h**, corrisponde l'indirizzo logico normalizzato **0DC9h:0000h**; la componente **0DC9h** di questo indirizzo, viene assegnata a **STACKSEGM**. Non appena inizia l'esecuzione del nostro programma, il valore **STACKSEGM** viene utilizzato quindi per inizializzare **SS**; infatti, in Figura 5 vediamo che **SS=0DC9h**.
Il registro **SP** viene inizializzato con il valore **0400h**, cosi' come avevamo richiesto nel listato di Figura 1; e' importante notare che in modalita' reale a **32** bit, i **16** bit piu' significativi di **ESP** valgono **0000h**.
Il fatto che il contenuto di **SP** resti costante nel corso dell'esecuzione del programma, indica che le varie procedure chiamate, gestiscono correttamente lo stack (nel senso che all'interno delle procedure, le istruzioni **PUSH** sono bilanciate perfettamente dalle istruzioni **POP**). Se, ad esempio, ci accorgiamo che **SP=0400h** prima della chiamata di una procedura, e **SP=03FCH** dopo la chiamata, allora c'e' sicuramente un errore di gestione dello stack all'interno della procedura stessa; ricordiamoci che in **Assembly**, lo strumento migliore per scovare gli errori e' il nostro cervello!
Per veder variare il contenuto di **SP**, possiamo scrivere istruzioni del tipo:

```
push     ax       ; salva AX nello stack
call     showCPU  ; chiama showCPU
pop      ax       ; ripristina AX
call     waitChar ; attende la pressione di un tasto
```

In questo caso, **showCPU** mostra per **SP** il valore **03FEh**; infatti, a causa dell'inserimento di una **WORD** nello stack, si ha:
`SP = 0400h - 0002h = 03FEh`
Subito dopo la chiamata di **showCPU**, l'istruzione **POP** riequilibra lo stack estraendo una **WORD** dall'indirizzo logico **SS:SP** che, in quel momento, vale **0DC9h:03FEh**.

In relazione ai vari flags, in Figura 5 notiamo, in particolare, che **PF=1** e **ZF=1**; cio' e' dovuto al fatto che prima di **showCPU**, viene chiamata la procedura **clearRegisters**, che azzera alcuni registri. In seguito all'azzeramento di un qualsiasi registro, si ottiene, ovviamente, **ZF=1**; ricordiamo, infatti, che **ZF** viene posto a livello logico **1**, ogni volta che una operazione produce un risultato pari a zero. I primi **8** bit del registro appena azzerato, valgono **00h**, e quindi contengono un numero pari (zero) di bit a livello logico **1** (lo zero viene trattato come numero pari); di conseguenza, il **parity flag PF** viene posto a livello logico **1**.

Nel corso dell'esecuzione del programma, vengono chiamate le procedure del tipo **readUdec**, **readSdec**, **readHex** e **readBin**, che richiedono all'utente, l'inserimento di numeri interi nelle tre ampiezze **8**, **16** e **32** bit, e nelle tre basi **10**, **16** e **2**; ogni numero inserito, viene poi visualizzato sullo schermo dalle procedure del tipo **writeUdec**, **writeSdec**, **writeHex** e **writeBin**.
Per ogni numero inserito, viene chiamata **showCPU** che mostra la nuova situazione dei registri della **CPU**; in particolare, si puo' notare che in **AL/AX/EAX** sono presenti i numeri appena inseriti, mentre in **DH** e **DL** sono presenti le coordinate di schermo che l'utente ha specificato per l'output.
Come e' stato detto in precedenza, il contenuto del registro **IP** si riferisce ai vari punti del programma, nei quali viene chiamata la procedura **showCPU**.

15.5 Programma di prova COMTEST2.ASM per la libreria COMLIB

Le considerazioni appena svolte, sono relative al collegamento di una libreria di procedure, ad un programma **Assembly** destinato ad essere convertito in un eseguibile in formato **EXE**; nel seguito del capitolo, esamineremo invece il caso del collegamento di una libreria di procedure, ad un programma **Assembly** destinato ad essere convertito in un eseguibile in formato **COM**.
Come gia' sappiamo, un programma in formato **COM** e' costituito da un unico segmento destinato a contenere, codice, dati e stack; questo significa che non possiamo utilizzare la libreria **EXELIB**, che contiene, al suo interno, due segmenti di programma distinti (uno per il codice e uno per i dati privati).
Proprio per questo motivo, si rende necessaria una versione modificata della libreria **EXELIB**, destinata esplicitamente ai programmi in formato **COM**; si tratta della libreria **COMLIB**, che, come e' stato gia' detto, contiene le stesse identiche procedure di **EXELIB**.
All'interno del file **COMLIB.OBJ**, tutto il codice e i dati si trovano inseriti all'interno di un unico segmento di programma, avente le seguenti caratteristiche:
`COMSEGM SEGMENT PARA PUBLIC USE16 'CODE'`
Tutti i programmi che vogliono fare uso di questa libreria, devono definire un segmento unico avente le stesse identiche caratteristiche di quello definito in **COMLIB**.

Per testare la libreria **COMLIB**, ci serviamo del file **COMTEST2.ASM**, illustrato in Figura 6; si tratta della versione in formato **COM**, del file **EXETEST2.ASM** di Figura 1.

Figura 6 - File COMTEST2.ASM

```
;--------------------------------------------------;
; File comtest2.asm                                ;
; Verifica a run-time di un programma COM          ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

INCLUDE     COMLIB.INC              ; inclusione libreria di I/O
.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

;################ segmento unico ##################

COMSEGM     SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM

   org      0100h                   ; libera 256 byte per il PSP

start:                              ; entry point

;------- inizio blocco principale istruzioni ------

   call     set80x50                ; modo video testo 80x50
   call     hideCursor              ; nasconde il cursore
   call     clearScreen             ; pulisce lo schermo

   call     clearRegisters          ; azzera i registri generali
   call     showCPU                 ; mostra lo stato della CPU

   mov      dx, 000eh               ; riga 00h, colonna 0Eh
   mov      di, offset strTitle     ; DI = Offset(strTitle)
```

```
call     writeString            ; stampa la stringa

call     readUdec8              ; legge un int. senza segno a 8 bit
mov      dx, 0300h              ; riga 03h, colonna 00h
call     writeUdec8             ; stampa il numero letto (AL)
call     showCPU                ; mostra lo stato della CPU

call     readUdec16             ; legge un int. senza segno a 16 bit
mov      dx, 0400h              ; riga 04h, colonna 00h
call     writeUdec16            ; stampa il numero letto (AX)
call     showCPU                ; mostra lo stato della CPU

call     readUdec32             ; legge un int. senza segno a 32 bit
mov      dx, 0500h              ; riga 05h, colonna 00h
call     writeUdec32            ; stampa il numero letto (EAX)
call     showCPU                ; mostra lo stato della CPU

call     clearRegisters         ; azzera i registri generali

call     readSdec8              ; legge un int. con segno a 8 bit
mov      dx, 0700h              ; riga 07h, colonna 00h
call     writeSdec8             ; stampa il numero letto (AL)
call     showCPU                ; mostra lo stato della CPU

call     readSdec16             ; legge un int. con segno a 16 bit
mov      dx, 0800h              ; riga 08h, colonna 00h
call     writeSdec16            ; stampa il numero letto (AX)
call     showCPU                ; mostra lo stato della CPU

call     readSdec32             ; legge un int. con segno a 32 bit
mov      dx, 0900h              ; riga 09h, colonna 00h
call     writeSdec32            ; stampa il numero letto (EAX)
call     showCPU                ; mostra lo stato della CPU

call     clearRegisters         ; azzera i registri generali

call     readHex8               ; legge un intero hex. a 8 bit
mov      dx, 0b00h              ; riga 0Bh, colonna 00h
call     writeHex8              ; stampa il numero letto (AL)
call     showCPU                ; mostra lo stato della CPU

call     readHex16              ; legge un intero hex. a 16 bit
mov      dx, 0c00h              ; riga 0Ch, colonna 00h
call     writeHex16             ; stampa il numero letto (AX)
call     showCPU                ; mostra lo stato della CPU

call     readHex32              ; legge un intero hex. a 32 bit
mov      dx, 0d00h              ; riga 0Dh, colonna 00h
call     writeHex32             ; stampa il numero letto (EAX)
call     showCPU                ; mostra lo stato della CPU

call     clearRegisters         ; azzera i registri generali

call     readBin8               ; legge un intero bin. a 8 bit
mov      dx, 0f00h              ; riga 0Fh, colonna 00h
call     writeBin8              ; stampa il numero letto (AL)
call     showCPU                ; mostra lo stato della CPU

call     readBin16              ; legge un intero bin. a 16 bit
mov      dx, 1000h              ; riga 10h, colonna 00h
call     writeBin16             ; stampa il numero letto (AX)
call     showCPU                ; mostra lo stato della CPU
```

```
   call     readBin32             ; legge un intero bin. a 32 bit
   mov      dx, 1100h             ; riga 11h, colonna 00h
   call     writeBin32            ; stampa il numero letto (EAX)
   call     showCPU               ; mostra lo stato della CPU

   mov      dx, 1400h             ; riga 14h, colonna 00h
   mov      di, offset strExit    ; ES:DI punta a strExit
   call     writeString           ; stampa la stringa

   call     waitChar              ; attende la pressione di un tasto
   call     clearScreen           ; pulisce lo schermo
   call     showCursor            ; ripristina il cursore

;-------- fine blocco principale istruzioni -------

   mov      ah, 4ch               ; servizio Terminate Program
   mov      al, 00h               ; exit code = 0
   int      21h                   ; chiama i servizi DOS

;----- inizio definizione variabili statiche ------

   ALIGN    4                     ; allinea alla DWORD
strTitle    db    'PROGRAMMA DI PROVA COMTEST2.ASM '
            db    'PER LA LIBRERIA COMLIB', 0
strExit     db    'Premere un tasto per terminare', 0

;------- fine definizione variabili statiche ------

COMSEGM     ENDS

;##################################################

   END      start
```

Notiamo subito che, all'interno di **COMTEST2.ASM**, e' presente un unico segmento di programma del tipo:

```
COMSEGM SEGMENT PARA PUBLIC USE16 'CODE'
```

Il nome e gli attributi di questo segmento, sono (e devono essere) identici a quelli dell'unico segmento di programma definito nel modulo **COMTEST.OBJ**.

Il listato del programma di Figura 6, e' assolutamente identico al listato del programma di Figura 1; l'unica importante eccezione, e' rappresentata dalla lista di argomenti che vengono passati alla procedura **writeString**.
Abbiamo visto che nel caso della libreria **EXELIB.OBJ**, la procedura **writeString** richiede l'indirizzo completo **Seg:Offset** (indirizzo **FAR**) della stringa da stampare; cio' accade in quanto **writeString**, non ha altro modo per conoscere il segmento di programma nel quale e' stata definita la stringa stessa.
Nel caso, invece, della libreria **COMLIB.OBJ**, la procedura **writeString** richiede la sola componente **Offset** (indirizzo **NEAR**) dell'indirizzo della stringa da stampare; cio' accade in quanto **writeString**, da' per scontato che tutti i dati del programma, siano stati definiti all'interno dell'unico blocco **COMSEGM**, e quindi, deduce che l'indirizzo logico completo della stringa da stampare, e' sicuramente **COMSEGM:Offset**.
Una diretta conseguenza della presenza di un unico segmento di programma, e' data dal fatto che, all'interno del file **COMLIB.INC**, troviamo dichiarazioni del tipo:

```
EXTRN set80x50: NEAR
```

In presenza del solo segmento **COMSEGM**, le varie procedure della libreria, vengono chiamate con una **NEAR call**; la **CPU** ha bisogno solamente di caricare in **IP** la componente **Offset** dell'indirizzo della procedura da chiamare, in quanto **CS** contiene gia' il valore **COMSEGM**.

15.5.1 Assembling & Linking del programma COMTEST2.ASM

La fase di assemblaggio del programma **COMTEST2.ASM**, e' del tutto simile al caso di **EXETEST2.ASM**; con il **TASM**, il comando da impartire e':
`..\bin\tasm /zn /l comtest2.asm`
Se vogliamo un assemblaggio **case-sensitive**, dobbiamo impartire il comando:
`..\bin\tasm /ml /zn /l comtest2.asm`

Con il **MASM**, il comando da impartire e':
`..\bin\ml /c /Fl comtest2.asm`
Se vogliamo un assemblaggio **case-sensitive**, dobbiamo impartire il comando:
`..\bin\ml /c /Cp /Fl comtest2.asm`

Premendo ora il tasto **[Invio]**, vengono generati i due file **COMTEST2.OBJ** e **COMTEST2.LST**; aprendo con un editor, il file **COMTEST2.LST**, possiamo constatare che, ad esempio, nel caso della chiamata alla procedura **set80x50**, l'assembler ha generato il codice macchina:
`E8h 0000h` (**NEAR call**).
Inoltre, nella **symbol table**, la procedura **set80x50** viene classificata dall'assembler come:
`set80x50 Near ----:---- Extern`
Come al solito, l'assembler sta delegando al linker, il compito di calcolare l'indirizzo logico **Seg:Offset** della procedura esterna **set80x50** di tipo **Near**.
In una situazione come quella di Figura 1, si puo' anche inserire la direttiva **INCLUDE**, all'interno del blocco **COMSEGM**; in tal caso, esaminando il file **COMTEST2.LST**, si nota che nella **symbol table**, l'assembler ha inserito informazioni del tipo:
`set80x50 Near COMSEGM:---- Extern`

Passiamo ora alla fase di linking del nostro programma; con il **TASM**, il comando da impartire e':
`..\bin\tlink /t comtest2.obj + comlib.obj`

Con il **MASM**, il comando da impartire e':
`..\bin\link /tiny /map comtest2.obj + comlib.obj`

Premendo il tasto **[Invio]**, parte la fase di linking che, in assenza di errori, porta alla generazione del file in formato eseguibile **COMTEST2.COM**, e del map file **COMTEST2.MAP**; analizziamo il lavoro svolto dal linker per il collegamento dei due moduli **COMTEST2.OBJ** e **COMLIB.OBJ**.

1) Il linker inizia ad esaminare il modulo **COMTEST2.OBJ**, dove trova il primo segmento **COMSEGM**.
2) Il linker cerca un eventuale altro segmento **COMSEGM** nel modulo **COMLIB.OBJ**, e dopo averlo trovato, lo unisce all'altro **COMSEGM**, ottenendo cosi' il blocco unico del nostro programma; questo blocco e' rappresentato dalla unione dei due **COMSEGM**, con allineamento **PARA**.
3) Non esistendo, ovviamente, altri segmenti, la fase di linking e' terminata.

Le considerazioni appena esposte, vengono confermate dal **map file** di Figura 7.

Figura 7 - COMTEST2.MAP

```
Start   Stop    Length  Name         Class

00000h  00FF8h  00FF9h  COMSEGM      CODE

Program entry point at 0000h:0100h
```

Ovviamente, l'**entry point** del nostro programma in formato **COM**, si trova all'indirizzo logico **0000h:0100h**.

Come si puo' facilmente immaginare, nel caso dei programmi in formato **COM**, diventa importante l'ordine con il quale si passano i vari moduli al linker; infatti, se proviamo ad eseguire il comando:
```
..\bin\tlink /t comlib.obj + comtest2.obj
```
otteniamo (con **TASM**) il messaggio di errore:
```
Fatal: Cannot generate COM file: data below initial CS:IP defined
```
Il linker ci sta dicendo che non puo' generare un eseguibile in formato **COM**, in quanto ha trovato definizioni di dati prima dell'**entry point**!
Il perche' di questo messaggio e' abbastanza chiaro; infatti, il linker inizia ad esaminare per primo il modulo **COMLIB.OBJ**, dove trova il segmento **COMSEGM**. A questo segmento, viene aggiunto l'altro **COMSEGM** presente in **COMTEST2.OBJ**; in questo modo, l'**entry point** del programma, non solo si viene a trovare ad un offset maggiore di **0100h**, ma viene preceduto dal codice e dai dati definiti nel modulo **COMLIB.OBJ**.

Nota importante.
Il **MASM**, permette la generazione di eseguibili in formato **COM**, il cui **entry point** puo' trovarsi in qualunque punto del segmento unico; di conseguenza, e' anche possibile creare il programma **COMLIB.COM**, attraverso il comando:
..\bin\ml /tiny /map comlib.obj + comtest2.obj
In casi di questo genere, il **MASM** si limita a mostrare il messaggio:
LINK warning: start address not equal to 0x0100 for /TINY
(**0x0100** e' la sintassi utilizzata dal linguaggio **C**, per indicare il numero esadecimale **0100h**).
E' ovvio che se si prova ad eseguire il programma **COMLIB.COM** cosi' ottenuto, si provoca un sicuro crash; molto spesso, questo tipo di crash puo' anche bloccare totalmente **Windows 98**!

Nota importante.
Il linker fornito con il **Borland TASM 5.0**, non permette la creazione di un programma in formato **COM** composto da due o piu' **object file**; questa limitazione non riguarda invece, ne' il linker del **MASM** (qualsiasi versione), ne' le precedenti versioni del linker della **Borland**.
Per superare questo problema quindi, bisogna per forza utilizzare il linker del **MASM** o il linker di una versione del **TASM** precedente la **5.0**; ricordando che i file in formato oggetto seguono lo standard **OMF**, si puo' utilizzare anche un qualunque altro linker per **DOS**, come quelli forniti, ad esempio, con il **Borland C/C++**, **Borland Pascal**, **Microsoft Fortran**, etc.

Esiste anche un metodo che permette ugualmente l'utilizzo del **TASM 5.0**, purche' si disponga del codice sorgente di tutti i moduli che compongono il programma da convertire in formato **COM**; supponiamo a tale proposito, di avere i due moduli, **MAIN.ASM** (programma principale) e **LIB1.ASM** (libreria di procedure). La tecnica da seguire, consiste innanzi tutto nell'eliminare la direttiva finale **END** dal modulo **LIB1.ASM**; a sua volta, il modulo **MAIN.ASM** deve assumere la seguente struttura:

```
COMLIB   SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM

   org      0100h

start:

........................

COMLIB   ENDS

INCLUDE  LIB1.ASM

   END   start
```

In questo modo, in fase di assemblaggio, il modulo **LIB1.ASM**, viene incorporato nel modulo **MAIN.ASM**; dopo l'assemblaggio, si ottiene un unico **object file**, chiamato **MAIN.OBJ**, che puo' essere passato al linker per poter ottenere l'eseguibile finale **MAIN.COM**.

15.5.2 Esecuzione del programma COMTEST2.COM

Passiamo ora alla fase di esecuzione del file **COMTEST2.COM**; posizionandoci nella cartella **ASMBASE**, dal **prompt** del **DOS** impartiamo il comando:

```
comtest2
```

e premiamo il tasto **[Invio]**.
Appena inizia l'esecuzione di **COMTEST2.COM**, vedremo comparire in fondo allo schermo, una serie di informazioni come quelle mostrate in Figura 8; queste informazioni, si riferiscono alla richiesta di input che arriva dalla procedura **readUdec8**, e allo stato della **CPU** mostrato dalla procedura **showCPU**.

Figura 8 - COMTEST2.COM in esecuzione
Inserire un numero in base 10 tra 0 e 255:
CPU
EAX=00000000H EBX=00000000H ECX=00000000H EDX=00000000H ESI=00000000H EDI=00000000H ESP=0000FFFEH EBP=00000912H CS=0CA5H DS=0CA5H ES=0CA5H FS=0000H GS=0000H SS=0CA5H IP=010CH CF = 0 PF = 1 AF = 0 ZF = 1 SF = 0 TF = 0 IF = 1 DF = 0 OF = 0

La procedura **showCPU**, chiamata proprio all'inizio del nostro programma, mostra il contenuto iniziale dei registri della **CPU**; analizziamo, in particolare, l'importante contenuto iniziale dei registri **CS**, **DS**, **ES**, **SS**, **SP** e **IP** (come e' stato gia' detto, le informazioni contenute nei registri di segmento, possono variare da computer a computer).

Come sappiamo, in un programma in formato **COM**, il **SO** inizializza i registri di segmento **CS**, **DS**, **ES** e **SS**, in modo che puntino tutti all'unico blocco presente; infatti, nel caso di Figura 8, vediamo che a questi quattro registri di segmento, e' stato assegnato lo stesso valore iniziale **0CA5h**.

In relazione al registro **IP**, il valore **010Ch** si riferisce, come al solito, al contenuto dell'**instruction pointer** al momento della prima chiamata di **showCPU**; infatti, analizzando il file **COMTEST2.LST**, si scopre che la prima chiamata di **showCPU**, si trova proprio all'indirizzo logico **COMSEGM:010Ch**.

La figura 8, conferma anche il fatto che in un programma in formato **COM**, il **SO** inizializza il registro **SP**, con il valore **FFFEh**; nel caso quindi del programma **COMTEST2.COM**, l'indirizzo logico iniziale della cima dello stack (**TOS**) e' **0CA5h:FFFEh**.

15.6 I batch file

In un precedente capitolo, e' stato detto che il **DOS** fornisce un particolare tipo di file eseguibile, chiamato **batch file**; si tratta di un normalissimo file di testo, contenente al suo interno una sequenza di comandi **DOS** eseguibili.
I **batch file** si rivelano molto utili nel momento in cui abbiamo bisogno di automatizzare alcune operazioni, particolarmente fastidiose; pensiamo, ad esempio, alla necessita' di riassemblare un intero programma formato da numerosi moduli.
Un **batch file**, puo' essere reso molto piu' flessibile, grazie alla possibilita' di utilizzare alcune particolari "pseudo-istruzioni", chiamate **comandi batch**; la Figura 9 illustra i **comandi batch** principali, e l'effetto prodotto da ciascuno di essi (per maggiori dettagli, si consiglia di consultare i manuali del **DOS**).

Figura 9 - Comandi batch

Comando	Effetto
REM	Delimita l'inizio di un commento che si sviluppa su una sola riga.
ECHO ON	Abilita la visualizzazione sullo schermo, dei comandi eseguiti dal DOS (opzione predefinita).
ECHO OFF	Disabilita la visualizzazione sullo schermo, dei comandi eseguiti dal DOS.
ECHO Stringa	Visualizza Stringa sullo schermo.
ECHO.	Visualizza una riga vuota sullo schermo.
PAUSE	Attende la pressione di un tasto.
IF EXIST NomeFile Comando Argomenti	Se NomeFile esiste, esegue Comando con Argomenti
IF NOT EXIST NomeFile Comando Argomenti	Se NomeFile non esiste, esegue Comando con Argomenti
GOTO Etichetta	Salta a Etichetta (definita nel batch file).
CALL NomeFile Argomenti	Chiama un'altro batch file NomeFile con Argomenti

Supponiamo, ad esempio, di voler automatizzare le fasi di assembling e linking del programma **EXETEST2.ASM**; a tale proposito, possiamo creare un apposito **batch file** chiamato **EXETEST2.BAT**. Il nome assegnato ad un **batch file**, deve avere, obbligatoriamente, l'estensione **BAT**; la Figura 10, illustra la struttura interna (volutamente contorta) di **EXETEST2.BAT**.

Figura 10 - EXETEST2.BAT

```
echo off
rem Assembling & Linking del programma exetest2.asm

cls
rem Se exetest2.obj e exetest2.exe esistono, li cancella
if not exist exetest2.obj goto delexe
echo Cancellazione di exetest2.obj
del exetest2.obj
echo.
:delexe
if not exist exetest2.exe goto assembla
echo Cancellazione di exetest2.exe
del exetest2.exe
echo.
:assembla
echo Assembling di exetest2.asm
..\bin\tasm /ml /zn /l exetest2.asm
echo.
echo Linking di exetest2.obj
..\bin\tasm exetest2.obj + exelib.obj
echo.
pause
cls
```

Come si puo' notare, le etichette devono iniziare con il simbolo **':'** (due punti); inoltre, e' importante osservare che l'istruzione da eseguire nel caso in cui sia verificata la condizione specificata da **IF**, deve trovarsi sulla stessa riga dello stesso **IF**.
Grazie al **batch file** di Figura 10, ogni volta che vogliamo ricreare il programma **EXETEST2.EXE** (ad esempio, dopo una modifica del codice sorgente), non dobbiamo fare altro che impartire il comando:
`exetest2.bat`
E' importante ricordare che, se si impartisce il comando **exetest2** senza specificare l'estensione, il **DOS** esegue il primo che incontra tra i file, **EXETEST2.COM**, **EXETEST2.EXE** e **EXETEST2.BAT**; proprio per questo motivo, nel nostro esempio dobbiamo necessariamente impartire il comando completo **exetest2.bat**, altrimenti, il **DOS** esegue **exetest2.exe** (se esiste).

Naturalmente, non siamo obbligati a scrivere un **batch file** complesso, come quello di Figura 10; possiamo anche limitarci a scrivere i soli comandi visibili in Figura 11.

Figura 11 - EXETEST2.BAT

```
..\bin\tasm /ml /zn /l exetest2.asm
..\bin\tasm exetest2.obj + exelib.obj
```

I **batch file**, sono in grado di gestire una serie di argomenti specificati dall'utente; all'interno di un **batch file**, i vari argomenti sono rappresentati, nell'ordine, dai simboli:
`%0, %1, %2, %3, %4, %5, %6, %7, %8, %9`
Il simbolo **%0** rappresenta il nome dello stesso **batch file**, per cui, il primo argomento passato dall'utente e' **%1**.

Tutto cio', ci permette di creare, ad esempio, un generico **batch file**, attraverso il quale possiamo linkare qualsiasi programma alla libreria **EXELIB.OBJ**; la Figura 12 illustra un esempio pratico, chiamato **EXELIB.BAT**, che rappresenta la generalizzazione del **batch file** di Figura 10.

Figura 12 - EXELIB.BAT

```
echo off
rem Assembling & Linking di un programma da collegare
rem alla libreria exelib.obj

cls
rem Se esistono, cancella il file.obj e il file.exe
if not exist %1.obj goto delexe
echo Cancellazione di %1.obj
del %1.obj
echo.
:delexe
if not exist %1.exe goto assembla
echo Cancellazione di %1.exe
del %1.exe
echo.
:assembla
echo Assembling di %1.asm
..\bin\tasm /ml /zn /l %1.asm
echo.
echo Linking di %1.obj
..\bin\tasm %1.obj + exelib.obj
echo.
pause
cls
```

Se, ad esempio, vogliamo creare il programma **EXETEST2.EXE**, non dobbiamo fare altro che impartire il comando:

```
exelib.bat exetest2
```

In questo caso, **exetest2** e' l'argomento che stiamo passando a **EXELIB.BAT**; e' fondamentale che l'argomento **exetest2**, venga passato senza l'estensione **asm**.

15.7 Esercitazioni consigliate

Si consiglia vivamente di sfruttare al massimo le librerie **EXELIB.OBJ** e **COMLIB.OBJ**, per effettuare il maggior numero possibile di esperimenti; solo in questo modo, si puo' acquisire la necessaria padronanza di tutti i concetti esposti nei precedenti capitoli.
Con le pochissime istruzioni (**MOV**, **ADD**, **PUSH**, **POP**) e con i pochissimi operatori (**SEG**, **OFFSET**, **BYTE PTR**, etc) che gia' conosciamo, possiamo scrivere un gran numero di programmi; nel seguito, vengono illustrati alcuni semplici esempi.

15.7.1 Visualizzazione dell'indirizzo logico di una informazione

Dopo aver definito i dati statici di un programma, possiamo provare a stampare sullo schermo gli indirizzi logici **Seg:Offset**, assegnati in fase di esecuzione ai dati stessi; supponendo, ad esempio, di aver definito un dato chiamato **Variabile1**, possiamo scrivere:

```
mov      dx, 0400h              ; riga 4, colonna 0
mov      ax, seg Variabile1     ; ax = seg Variabile1
call     writeHex16             ; stampa ax
add      dh, 1                  ; incrementa la riga
mov      ax, offset Variabile1  ; ax = offset Variabile1
call     writeHex16             ; stampa ax
```

Al posto di **Variabile1**, possiamo utilizzare anche il nome di una etichetta o di una procedura, definite in un blocco di codice; ad esempio, si puo' provare a visualizzare l'indirizzo **Seg:Offset** di una qualsiasi procedura definita nella libreria **EXELIB.OBJ** o **COMLIB.OBJ**.
Ricordiamoci che in un eseguibile in formato **COM**, e' proibito scrivere istruzioni del tipo:

```
mov ax, seg writeString
```

Al posto di questa istruzione, possiamo scrivere, ad esempio:

```
mov ax, cs
```

Infatti, in un programma in formato **COM**, e' presente un unico segmento, referenziato sicuramente da **CS**; anche gli altri tre registri **DS**, **ES** e **SS**, se non hanno subito modifiche, referenziano l'unico segmento di programma presente.

15.7.2 Gestione di un dato attraverso un puntatore NEAR o FAR

Dopo aver definito i dati statici di un programma, possiamo provare a gestirli attraverso un puntatore **NEAR** o **FAR**; supponiamo, ad esempio, di aver definito in un blocco **DATASEGM**, il dato:

```
Variabile1 dw -14532
```

Dopo aver caricato **DATASEGM** in **DS**, possiamo scrivere:

```
mov      ax, seg Variabile1      ; ax = seg Variabile1
mov      es, ax                  ; es = ax
mov      di, offset Variabile1   ; di = offset Variabile1

; Gestione di Variabile1 con il puntatore NEAR DI
; La CPU associa automaticamente DI a DS, per cui
; non e' necessario il segment override DS:[DI]

mov      dx, 0400h               ; riga 4, colonna 0
mov      ax, [di]                ; ax = ds:[di] = -14532
call     writeSdec16             ; stampa ax

; Gestione di Variabile1 con il puntatore FAR ES:DI
; La CPU non associa automaticamente DI a ES, per cui
; e' necessario il segment override ES:[DI]

add      dh, 1                   ; incrementa la riga
mov      ax, es:[di]             ; ax = es:[di] = -14532
call     writeSdec16             ; stampa ax
```

E' importante ribadire che prima di utilizzare **DI** come puntatore **NEAR**, il registro di segmento **DS** deve essere gia' stato inizializzato con **DATASEGM**; in caso contrario, il simbolo **[DI]** fornisce un valore privo di senso, dovuto al fatto che **DS** puo' avere un contenuto casuale.

15.7.3 Accesso allo stack con SS:BP

Come sappiamo, in modalita' reale e' proibito dereferenziare il registro **SP** attraverso istruzioni del tipo:
```
mov ax, [sp] ; ax = ss:[sp]
```
Al posto di **SP** dobbiamo allora utilizzare **BP**; se vogliamo esplorare lo stack con **BP**, possiamo scrivere, ad esempio:

```
mov      ax, 45920                ; ax = 45920
mov      bx, 52140                ; bx = 52140
push     ax                       ; salva 45920 nello stack
push     bx                       ; salva 52140 nello stack

; Il valore 52140 si trova nello stack all'indirizzo SS:SP (TOS)
; Il valore 45920 si trova nello stack all'indirizzo SS:(SP+2)
; La CPU associa automaticamente SP e BP a SS, per cui
; non e' necessario il segment override SS:[SP] o SS:[BP]

mov      bp, sp                   ; ss:bp = ss:sp
mov      dx, 0400h                ; riga 4, colonna 0
mov      ax, [bp]                 ; ax = ss:[bp] = 52140
call     writeUdec16              ; stampa ax

add      dh, 1                    ; incrementa la riga
mov      ax, [bp+2]               ; ax = ss:[bp+2] = 45920
call     writeUdec16              ; stampa ax

pop      bx                       ; bx = 52140
pop      ax                       ; ax = 45920

; Possiamo anche usare una sola istruzione POP, con operando EAX,
; in modo da estrarre contemporaneamente 32 bit dallo stack.
```

Ricordiamo che ogni istruzione **PUSH** con operando di tipo **WORD**, decrementa **SP** di **2**, per cui le varie **PUSH**, inseriscono i loro operandi ad indirizzi sempre piu' bassi dello stack (riempimento dello stack); come si nota nella parte finale del precedente listato, se vogliamo ripristinare esattamente il vecchio contenuto di **AX** e **BX**, dobbiamo estrarre gli operandi dallo stack, in senso inverso rispetto agli inserimenti (infatti, lo stack e' una struttura di tipo **LIFO**).
Naturalmente, nessuno ci impedisce di accedere allo stack con, ad esempio, **ES:SI**; in questo caso, dobbiamo caricare **SP** in **SI** e **SS** in **ES**. L'utilizzo di **ES:SI** richiede il segment override, per cui, dobbiamo scrivere istruzioni del tipo:
```
mov ax, es:[si+2]
```

15.7.4 Visualizzazione dei codici ASCII contenuti in una stringa

Supponiamo di aver definito in un blocco **DATASEGM**, la stringa:
`Messaggio db 'Assembly Programming'`
Se vogliamo visualizzare i codici ASCII dei vari simboli che formano la stringa, possiamo scrivere:

```
mov      dx, 0400h              ; riga 4, colonna 0
mov      al, Messaggio[0]       ; al = ds:Messaggio[0] = 'A'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
mov      al, Messaggio[1]       ; al = ds:Messaggio[1] = 's'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
mov      al, Messaggio[2]       ; al = ds:Messaggio[2] = 's'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
mov      al, Messaggio[3]       ; al = ds:Messaggio[3] = 'e'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
; e cosi' via
```

Questo esempio funziona solo se **DATASEGM** e' stato gia' caricato in **DS**, e se e' presente una direttiva **ASSUME** che associa **DATASEGM** a **DS**; in assenza della direttiva **ASSUME**, dobbiamo scrivere, esplicitamente, **DS:Messaggio[0]**, **DS:Messaggio[1]**, etc.
Volendo utilizzare i registri puntatori, possiamo scrivere:

```
mov      dx, 0400h              ; riga 4, colonna 0
mov      bx, offset Messaggio   ; bx = offset Messaggio

; La CPU associa automaticamente BX a DS, per cui
; non e' necessario il segment override DS:[BX]

mov      al, [bx+0]             ; al = ds:[bx+0] = 'A'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
mov      al, [bx+1]             ; al = ds:[bx+1] = 's'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
mov      al, [bx+2]             ; al = ds:[bx+2] = 's'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
mov      al, [bx+3]             ; al = ds:[bx+3] = 'e'
call     writeHex8              ; stampa al
add      dh, 1                  ; incrementa la riga
; e cosi' via
```

Questo esempio funziona solo se **DATASEGM** e' stato gia' caricato in **DS**; la direttiva **ASSUME** che associa **DATASEGM** a **DS** e' superflua in quanto stiamo utilizzando, esplicitamente, **DS:BX** (cioe', **DATASEGM:BX**) per l'accesso ai dati.
Volendo utilizzare **BP** per indirizzare **Messaggio**, dobbiamo scrivere, esplicitamente, **DS:[BP+0]**, **DS:[BP+1]**, etc; in caso contrario, la **CPU** associa automaticamente **BP** a **SS**.

15.7.5 Analisi del risultato di una operazione attraverso i flags

Sfruttando la procedura **showCPU**, possiamo verificare in pratica tutti i concetti sulla matematica del computer, esposti nei precedenti capitoli; per il momento, dobbiamo limitarci a degli esempi che utilizzano la sola istruzione **ADD**.

Se vogliamo provocare un **Carry** dal nibble basso al nibble alto di **AL**, possiamo scrivere:

```
call     clearRegisters          ; azzera i registri generali
mov      al, 00001111b           ; al = 0Fh
add      al, 1                   ; al = 0Fh + 01h = 10h
call     showCPU                 ; mostra AF = 1
call     waitChar                ; premere un tasto
```

Sommando **1** al valore binario **00001111b**, provochiamo un riporto dal nibble basso al nibble alto di **AL**; di conseguenza, il flag **AF** (**Auxiliary Carry Flag**) si porta a livello logico **1**.

Se vogliamo provocare un **Carry** in una somma con **AX**, possiamo scrivere:

```
call     clearRegisters          ; azzera i registri generali
mov      ax, 65535               ; ax = FFFFh
add      ax, 1                   ; ax = FFFFh + 0001h = 0000h
call     showCPU                 ; mostra CF = 1
call     waitChar                ; premere un tasto
```

Sommando **1** al valore **65535**, otteniamo un valore con ampiezza maggiore di **16** bit, che provoca quindi un riporto; di conseguenza, il flag **CF** (**Carry Flag**) si porta a livello logico **1**.

Se vogliamo provocare un **Overflow** in una somma con **AX**, possiamo scrivere:

```
call     clearRegisters          ; azzera i registri generali
mov      ax, +32767              ; ax = 7FFFh
add      ax, 1                   ; ax = 7FFFh + 0001h = 8000h
call     showCPU                 ; mostra OF = 1 e SF = 1
call     waitChar                ; premere un tasto
```

Sommando **1** al valore **+32767**, otteniamo **32768** che, per i numeri interi senza segno, rappresenta **32768**, mentre per i numeri interi con segno, rappresenta **-32768**; di conseguenza, il flag **CF** (**Carry Flag**) si porta a livello logico **0**, mentre i flag **OF** (**Overflow Flag**) e **SF** (**Sign Flag**) si portano entrambi a livello logico **1**.

15.7.6 Visualizzazione del campo CommandLineParameters del PSP di un programma

Utilizzando le informazioni presenti nella Figura 11 del Capitolo 13, possiamo provare a visualizzare la stringa dei parametri che un programma accetta dalla linea di comando; a tale proposito, dobbiamo tenere presente che il campo **CommandLineParmLength** occupa **1** byte, e si trova all'indirizzo logico **PSP:0080h**, mentre il campo **CommandLineParameters**, occupa sino a **127** byte, e si trova all'indirizzo logico **PSP:0081h**.
Non appena un programma (**COM** o **EXE**) viene caricato in memoria, sappiamo che **ES=PSP**; di conseguenza, prima di chiamare **writeString**, dobbiamo solo caricare in **DI** l'offset **0081h**. Bisogna anche ricordare che la procedura **writeString**, richiede una stringa **C** terminata da un byte di valore **0**; questo byte, lo dobbiamo mettere alla fine della stringa **CommandLineParameters**, e cioe', nella posizione che si ricava da **ES:[0080h]**.
In base a queste considerazioni, possiamo scrivere:

```
mov      di, 0081h                ; es:di punta alla stringa dei parametri
mov      bl, es:[0080h]           ; bl = lunghezza stringa
mov      bh, 0                    ; bh = 0
mov      byte ptr es:[bx+di], 0   ; mette uno zero alla fine della stringa
mov      dx, 0400h                ; riga 4, colonna 0
call     writeString              ; visualizza la stringa
```

Supponiamo che il nome di questo programma sia **progtest.exe**, e proviamo ad eseguire il comando:

```
progtest mela pera ciliegia
```

La stringa dei parametri e' lunga **19** byte, compresi gli spazi; otteniamo quindi all'indirizzo **ES:[0081h]**:

```
CommandLineParameters db ' mela pera ciliegia', 0Dh
```

e all'indirizzo **ES:[0080h]**:

```
CommandLineParmLength db 19
```

Caricando **0081h** in **DI**, facciamo puntare **ES:DI** alla stringa dei parametri; a questo punto, dobbiamo mettere un byte di valore zero in posizione **19** della stessa stringa. A tale proposito, carichiamo il byte di valore **19** in **BL** con l'istruzione:

```
mov bl, es:[0080h]
```

Carichiamo ora il valore zero in **BH** in modo da avere **BX=19**; a questo punto, possiamo inserire in **CommandLineParameters** il byte di valore zero, con l'istruzione:

```
mov byte ptr es:[bx+di], 0
```

Osserviamo che:

```
BX + DI = 19 + 0081h = 0013h + 0081h = 0094h
```

A questo punto, la stringa e' pronta per essere visualizzata da **writeString**.

A partire dal prossimo capitolo, verranno illustrate in dettaglio, tutte le istruzioni delle **CPU 80x86**, destinate alla modalita' reale; in questo modo, sara' possibile presentare programmi di esempio, molto piu' complessi rispetto a quelli presentati in questo capitolo.

Capitolo 16 - Istruzioni per il trasferimento dati

Come e' stato spiegato nei precedenti capitoli, ogni famiglia di **CPU** mette a disposizione un vasto numero di istruzioni che, nel loro insieme, formano il cosiddetto set di istruzioni della CPU; in base al compito che devono svolgere, le istruzioni possono essere suddivise in varie categorie. Si possono citare, ad esempio, le istruzioni per il trasferimento dati, le istruzioni aritmetiche, le istruzioni logiche, le istruzioni per le stringhe, le istruzioni per il controllo della **CPU** e cosi' via; in questo capitolo vengono esaminate in dettaglio, le principali istruzioni destinate al trasferimento dati da una sorgente a una destinazione.

Per ricavare i codici macchina di tutte le istruzioni presentate in questo capitolo e nei capitoli successivi, sono stati utilizzati i documenti ufficiali della **Intel**, denominati **231455.PDF** (dal titolo **Intel 8086 16 BIT HMOS MICROPROCESSOR**) e **24319101.PDF** (dal titolo **Intel Architecture Software Developer's Manual - Volume 2 - Instruction Set Reference**); questi documenti (in formato **PDF**), possono essere scaricati liberamente, dal sito ufficiale della **Intel**.

Ovviamente, tutte le istruzioni che coinvolgono operandi a **32** bit, presuppongono la presenza di una **CPU 80386** o superiore; questo aspetto viene dato per scontato nel seguito del capitolo e nei capitoli successivi (per maggiori dettagli, si puo' consultare la tabella delle istruzioni della CPU).

Nota importante.
Come e' stato gia' spiegato nei precedenti capitoli, l'assembler utilizza l'attributo **Dimensione** di un segmento di programma, per stabilire automaticamente la modalita' di indirizzamento predefinita; nel seguito del capitolo e nei capitoli successivi, si assume che questo attributo sia sempre di tipo **USE16**. In questo modo, l'assembler assegna ad ogni informazione, un indirizzo logico **Seg:Offset** con componente **Seg** a **16** bit, e con componente **Offset** a **16** bit; se si utilizzano esplicitamente delle componenti **Offset** a **32** bit, e' importante che la loro **WORD** piu' significativa valga **0000h**. Nel caso, ad esempio, di una istruzione come:

```
mov ax, ds:[ebx]
```

l'offset contenuto in **EBX** non deve superare il valore **0000FFFFh**.

16.1 L'istruzione MOV

Come si puo' facilmente intuire, l'istruzione **MOV** e' sicuramente quella che viene utilizzata in modo piu' massiccio nei programmi **Assembly**; attraverso questo mnemonico, si indica l'istruzione **MOVe data** (trasferimento dati), che ha lo scopo di copiare una informazione, da un operando sorgente (**SRC**) ad un operando destinazione (**DEST**).
La copia avviene bit per bit, nel senso che:

***** il bit in posizione **0** di **SRC** viene copiato nel bit in posizione **0** di **DEST**,
***** il bit in posizione **1** di **SRC** viene copiato nel bit in posizione **1** di **DEST**,
***** il bit in posizione **2** di **SRC** viene copiato nel bit in posizione **2** di **DEST**,
e cosi' via.

Tutto cio' implica che **SRC** e **DEST** debbano avere la stessa identica ampiezza in bit; in caso contrario, l'assembler genera un messaggio di errore.

Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
MOV   Reg, Reg
MOV   Reg16, SegReg
MOV   SegReg, Reg16
MOV   Mem, Reg
MOV   Reg, Mem
MOV   Mem16, SegReg
MOV   SegReg, Mem16
MOV   Reg, Imm
MOV   Mem, Imm
```

Come sappiamo, le **CPU** della famiglia **80x86** non permettono il trasferimento dati (o qualsiasi altra operazione) tra **Mem** e **Mem**; nel caso particolare del trasferimento dati, ci si puo' servire della tecnica del **DMA** che verra' analizzata nella sezione **Assembly Avanzato**.
E' illegale anche il trasferimento dati da **SegReg** a **SegReg** e da **Imm** a **SegReg**; ovviamente, in tutte le istruzioni che coinvolgono registri di segmento, gli operandi sono necessariamente a **16** bit.
Il trasferimento dati (o qualsiasi altra operazione) verso una destinazione di tipo **Imm**, e' chiaramente una istruzione priva di senso; infatti, sappiamo che un operando di tipo **Imm** e' un semplice valore numerico che non e' contenuto, ne' in un registro, ne' in una locazione di memoria. Cio' impedisce alla **CPU** di accedere in scrittura a questo tipo di operando, per modificarne il contenuto; nei precedenti capitoli abbiamo visto che un **Imm** che figura come operando **SRC** di una istruzione, viene incorporato dall'assembler, direttamente nel codice macchina dell'istruzione stessa.
Nota Importante.
In base a quanto e' stato appena detto, e' permesso il trasferimento dati da **Reg16/Mem16** a **SegReg**; l'unica comprensibile eccezione, e' rappresentata da un trasferimento dati verso la destinazione **CS**, come ad esempio:

```
MOV CS, AX
```

Solamente la **CPU** (nell'esecuzione delle istruzioni di salto), puo' modificare direttamente il contenuto di **CS**; naturalmente, un esperto programmatore **Assembly**, puo' facilmente accedere in modo indiretto a **CS**, modificandone il contenuto. Appare chiaro il fatto che, inserendo in **CS** valori privi di senso, si provoca l'immediato crash del programma in esecuzione!

In presenza di un trasferimento dati con destinazione **SS** (o di una qualsiasi modifica del registro **SS**), la **CPU** pone automaticamente a **0** il flag **IF** (**interrupt enable flag**); dopo l'esecuzione dell'istruzione successiva (che spesso e' un'altra **MOV** con destinazione **SP**), la **CPU** riporta automaticamente a **1** il flag **IF**.
Tutto cio' avviene in quanto, come e' stato gia' spiegato in un precedente capitolo, tutte le operazioni di "personalizzazione" dello stack devono svolgersi con le interruzioni mascherabili disabilitate; a causa di un bug, le vecchie **CPU 8086** non disabilitavano **IF** durante la modifica di **SS**. In generale, per evitare sorprese durante la modifica della coppia **SS:SP**, e' consigliabile servirsi delle istruzioni **CLI** e **STI**, in modo da provvedere personalmente al controllo delle interruzioni mascherabili.

Molte delle considerazioni che vengono esposte in questo capitolo in relazione alla istruzione **MOV**, valgono nel caso generale, per tutte le istruzioni della **CPU**; di conseguenza, l'istruzione **MOV** verra' analizzata con estremo dettaglio, in modo da non dover ripetere per le altre istruzioni, tutti questi concetti di validita' generale.

L'aspetto che crea piu' (ingiustificati) problemi ai programmatori **Assembly** meno esperti, e' dato dalla corretta gestione degli indirizzamenti relativi ad eventuali operandi di tipo **Mem** presenti nelle istruzioni; sfruttiamo allora proprio l'istruzione **MOV**, per analizzare i vari casi che si possono presentare.

A tale proposito, consideriamo un programma **Assembly** in formato **EXE**, dotato di, un segmento di dati chiamato **DATASEGM**, un segmento di codice chiamato **CODESEGM** e un segmento di stack chiamato **STACKSEGM**; vediamo allora quello che succede quando una istruzione, coinvolge un operando di tipo **Mem**, cioe' un dato statico, definito in uno qualsiasi dei tre segmenti appena citati.
Negli esempi che seguono, si assume per semplicita', che la componente **Offset** del dato non subisca nessuna rilocazione, e che il segmento di programma nel quale viene definito il dato stesso, sia allineato al paragrafo; inoltre, assumiamo che il dato sia correttamente allineato in memoria, in modo che la **CPU** lo possa leggere o scrivere, con un unico accesso.

16.1.1 Il dato si trova in DATASEGM

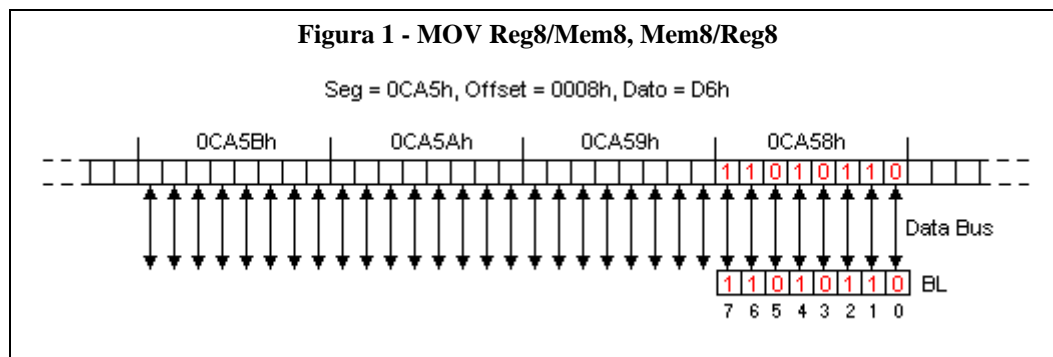
Supponiamo che all'offset **0008h** del blocco **DATASEGM** del nostro programma, sia presente la seguente definizione:

```
Variabile8 db 0d6h ; 11010110b
```

Quando l'assembler incontra questa definizione, crea all'offset **0008h** di **DATASEGM**, una locazione da **8** bit, nella quale inserisce il valore iniziale **D6h**, cioe' **11010110b**.
Supponiamo inoltre che quando inizia l'esecuzione del programma, si abbia:

```
DATASEGM = 0CA5h
```

Di conseguenza, il dato **Variabile8** viene caricato in memoria, all'indirizzo logico **DATASEGM:0008h**, cioe' **0CA5h:0008h**.
Ad un certo punto del nostro programma, abbiamo la necessita' di effettuare un trasferimento dati da **Variabile8** al registro **BL**; analizziamo allora i vari metodi di indirizzamento che ci permettono di svolgere questa operazione. La Figura 1 illustra in modo schematico, come avviene il trasferimento dati da **Variabile8** a **BL** e viceversa, attraverso un **Data Bus** a **32** bit.

Figura 1 - MOV Reg8/Mem8, Mem8/Reg8

La Figura 1 ci permette di sottolineare ancora una volta, che conviene sempre disegnare il vettore delle celle di memoria, con gli indirizzi crescenti da destra verso sinistra; in questo modo, tutti i dati di tipo **WORD**, **DWORD**, etc, presenti in memoria, appaiono disposti nel verso previsto dalla notazione posizionale (cioe', con il peso delle cifre che cresce da destra verso sinistra).

Normalmente, subito dopo l'**entry point** del programma, sono presenti le classiche istruzioni che caricano **DATASEGM** in **DS**; inoltre, e' presente anche una direttiva **ASSUME** che associa **DATASEGM** a **DS**.
Questa situazione, ci permette di gestire **Variabile8** nel modo piu' semplice possibile; infatti, sempre in riferimento alla Figura 1, possiamo scrivere l'istruzione:

```
mov bl, Variabile8
```

Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e da un **Disp16**; nel nostro caso, l'istruzione e' **to register** (**d=1**), e gli operandi sono a **8** bit (**w=0**). Otteniamo quindi:

```
Opcode = 10001010b = 8Ah
```

Il registro destinazione e' **BL**, per cui (Capitolo 11) **reg=011b**; la sorgente e' un **Disp16**, per cui **mod=00b**, **r/m=110b**. Otteniamo allora:
```
mod_reg_r/m = 00011110b = 1Eh
```
Il **Disp16** e':
```
Disp16 = 0000000000001000b = 0008h
```
Grazie alla direttiva **ASSUME**, l'assembler tratta **Variabile8** come un **Disp16** riferito a **DS**; non essendo necessario nessun segment override (in quanto **DS** e' il registro di segmento predefinito per i dati), l'assembler genera quindi il codice macchina:
```
10001010b 00011110b 0000000000001000b = 8Ah 1Eh 0008h
```
Quando la **CPU** incontra questa istruzione, esegue le seguenti operazioni:

1) L'offset **0008h** viene associato automaticamente a **DS**.
2) L'indirizzo logico **DS:0008h** (cioe' **0CA5h:0008h**) viene inviato al **MAR** (**Memory Address Register**).
3) Il **MAR** converte l'indirizzo logico **0CA5h:0008h**, nell'indirizzo fisico:
```
(0CA5h * 10h) + 0008h = 0CA50h + 0008h = 0CA58h
```
e lo carica sull'**Address Bus**.
4) L'**Address Bus** seleziona la cella di memoria n. **0CA58h**, e la mette in comunicazione con la **CPU**.
5) Attraverso le linee da **D0** a **D7** del **Data Bus**, la **CPU** legge dalla memoria gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**.

Cosa succede se non e' presente la direttiva **ASSUME**?
In tal caso, siamo costretti a scrivere in modo esplicito:
```
mov bl, ds:Variabile8
```
Incontrando questa istruzione, l'assembler genera lo stesso identico codice macchina precedente, in quanto **DS** e' il registro di segmento predefinito per i dati; ovviamente, l'assembler sa' che la **CPU**, associa automaticamente l'offset **0008h**, alla componente **Seg** contenuta in **DS**.

Negli esempi appena presentati, il dato **Variabile8** e' stato gestito attraverso un indirizzo **NEAR**; infatti, nel codice macchina viene inserita la sola componente **Offset** di **Variabile8**.

Supponiamo ora di voler gestire **DATASEGM** con **ES**; a tale proposito, carichiamo **DATASEGM** in **ES** e utilizziamo una direttiva **ASSUME** per associare **DATASEGM** ad **ES**.
In base a queste premesse, analizziamo quello che accade quando l'assembler incontra la solita istruzione:
```
mov bl, Variabile8
```
Grazie alla precedente direttiva **ASSUME**, l'assembler tratta il nome **Variabile8** come un **Disp16** riferito a **ES**; questo registro di segmento non e' quello predefinito per i dati, e quindi e' necessario un segment override. L'assembler genera un codice macchina del tutto simile a quello del precedente esempio, preceduto pero' dal codice **26h**, relativo al registro **ES**; si ottiene quindi:
```
00100110b 10001010b 00011110b 0000000000001000b = 26h 8Ah 1Eh 0008h
```
Quando la **CPU** incontra questa istruzione, associa l'offset **0008h** ad **ES**, accede in memoria all'indirizzo **ES:0008h** (cioe' a **0CA5h:0008h**), legge gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**; in assenza del codice **26h**, la **CPU** avrebbe associato l'offset **0008h** a **DS**.

Come al solito, se non e' presente la direttiva **ASSUME** che associa **DATASEGM** a **ES**, siamo costretti a scrivere in modo esplicito:
```
mov bl, es:Variabile8
```
Il codice macchina che si ottiene e' identico a quello precedente.

L'utilizzo di **ES**, che non e' il registro di segmento predefinito per i dati, comporta la gestione di **Variabile8** attraverso un indirizzo di tipo **FAR**, cioe', un indirizzo logico completo **Seg:Offset**;

ogni riferimento a **Variabile8** in una istruzione, costringe l'assembler a generare un codice macchina preceduto da un segment override.
Un segment override inserito in una istruzione, fa' crescere di **1** byte la dimensione del codice macchina dell'istruzione stessa; di conseguenza, una istruzione contenente un segment override, occupa maggiore spazio in memoria, e viene elaborata meno velocemente dalla **CPU**. L'ideale sarebbe quindi utilizzare sempre indirizzamenti di tipo **NEAR**; come vedremo pero' nel seguito del capitolo e nei capitoli successivi, in molti casi non esiste alternativa all'utilizzo degli indirizzamenti di tipo **FAR**.

Come si puo' facilmente intuire, se carichiamo **DATASEGM**, sia in **DS**, sia in **ES**, e poi scriviamo:

```
ASSUME DS: DATASEGM, ES:DATASEGM
```

allora l'assembler non inserira' nessun segment override nel codice macchina dell'istruzione:

```
mov bl, Variabile8
```

Infatti, in un caso di questo genere, l'assembler sfrutta automaticamente l'associazione tra **DS** e **DATASEGM**.

Passiamo ora alla gestione di **Variabile8** attraverso i registri puntatori; in questo caso, la direttiva **ASSUME** diventa del tutto superflua in quanto stiamo indicando esplicitamente all'assembler (e quindi anche alla **CPU**) i registri da utilizzare per indirizzare **Variabile8**.
Per evitare errori grossolani, e' importante ricordare che, tra registri puntatori e registri di segmento, in assenza di segment override valgono le seguenti associazioni predefinite:

*** BX**, **SI** e **DI** vengono associati automaticamente a **DS**;
*** BP** e **SP** vengono associati automaticamente a **SS**;
*** IP** viene associato automaticamente a **CS**.

Negli **effective address** del tipo **[base+index+disp]**, in assenza di segment override, se la **base** e' **BP** viene utilizzato il registro di segmento predefinito **SS**; se la **base** e' **BX**, il registro di segmento predefinito e' **DS**.
Nota importante.
Quando si lavora in modalita' reale, e' consigliabile evitare l'uso dei registri puntatori a **32** bit; in questo modo, si ottengono programmi piu' semplici da scrivere e da capire. In presenza di segmenti di programma con attributo **USE16**, l'uso dei registri puntatori a **32** bit non porta nessun beneficio; esiste anche il rischio di provocare un crash del programma, a causa dell'utilizzo involontario di un offset a **32** bit, maggiore di **0000FFFFh**.

Supponiamo allora di voler gestire **Variabile8** attraverso il registro puntatore **DI**; carichiamo innanzi tutto **DATASEGM** in **DS**, e scriviamo poi:

```
mov di, offset Variabile8
```

Questa istruzione carica il valore **0008h** in **DI**.
In base a queste premesse, possiamo scrivere l'istruzione:

```
mov bl, [di]
```

Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m**; nel nostro caso, l'istruzione e' **to register**, per cui **d=1**. Grazie a **BL**, l'assembler capisce che gli operandi sono a **8** bit (**w=0**); otteniamo quindi:

```
Opcode = 10001010b = 8Ah
```

Il registro destinazione e' **BL**, per cui **reg=011b**; la sorgente e' un **effective address** del tipo **[DI]**, per cui **mod=00b**, **r/m=101b**. Otteniamo allora:

```
mod_reg_r/m = 00011101b = 1Dh
```

In mancanza di diverse indicazioni da parte del programmatore, il registro puntatore **DI** viene associato automaticamente a **DS**, per cui non e' necessario nessun segment override; l'assembler genera quindi il codice macchina:

```
10001010b 00011101b = 8Ah 1Dh
```

Quando la **CPU** incontra questa istruzione, legge l'offset **0008h** contenuto in **DI**, accede in memoria all'indirizzo **DS:0008h** (cioe' a **0CA5h:0008h**), legge gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**.

Nell'esempio appena illustrato, il dato **Variabile8** e' stato gestito con un indirizzo **NEAR** contenuto in **DI**; cio' e' una diretta conseguenza della associazione predefinita tra **DI** e **DS**.

Se abbiamo caricato **DATASEGM** in **ES**, la precedente istruzione deve essere scritta, necessariamente, come:

```
mov bl, es:[di]
```

Il registro **DI**, non viene associato automaticamente a **ES**, per cui l'assembler, genera un codice macchina simile al precedente, preceduto pero' dal segment override **26h** relativo allo stesso **ES**; in caso contrario, **DI** verrebbe associato automaticamente a **DS**. Si ottiene quindi:

```
00100110b 10001010b 00011101b = 26h 8Ah 1Dh
```

Quando la **CPU** incontra questa istruzione, legge l'offset **0008h** contenuto in **DI**, accede in memoria all'indirizzo **ES:0008h** (cioe' a **0CA5h:0008h**), legge gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**; in assenza del codice **26h**, la **CPU** avrebbe associato l'offset **0008h** al registro **DS**.

Nell'esempio appena illustrato, il dato **Variabile8** e' stato gestito con un indirizzo **FAR** contenuto in **ES:DI**; cio' e' una diretta conseguenza del fatto che **DI**, non viene associato automaticamente a **ES**.

Se vogliamo complicarci la vita, nessuno ci impedisce di accedere a **Variabile8** con **BP**; in questo caso, dobbiamo ricordarci che **BP** viene associato automaticamente a **SS**, per cui dobbiamo ricorrere, in ogni caso, al segment override.
Anche se abbiamo caricato **DATASEGM** in **DS**, dobbiamo scrivere quindi:

```
mov bl, ds:[bp]
```

Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m**; nel nostro caso, l'istruzione e' **to register**, per cui **d=1**. Grazie a **BL**, l'assembler capisce che gli operandi sono a **8** bit (**w=0**); otteniamo quindi:

```
Opcode = 10001010b = 8Ah
```

Il registro destinazione e' **BL**, per cui **reg=011b**; la sorgente e' un **effective address** del tipo **[BP]**. Come abbiamo visto nel Capitolo 11, in questo caso si usa la forma **[BP+Disp8]** aggiungendo al codice macchina un **Disp8=00h** (spiazzamento a **8** bit); la codifica di questo tipo di indirizzamento e' **mod=01b**, **r/m=110b**. Otteniamo allora:

```
mod_reg_r/m = 01011110b = 5Eh
```

Il **Disp8** e':

```
Disp8 = 00000000b = 00h
```

Il registro **BP**, non viene associato automaticamente a **DS**, per cui l'assembler, genera un codice macchina preceduto dal segment override **3Eh** relativo allo stesso **DS**; in caso contrario, **BP** verrebbe associato automaticamente a **SS**. Si ottiene quindi:

```
00111110b 10001010b 01011110b 00000000b = 3Eh 8Ah 5Eh 00h
```

Quando la **CPU** incontra questa istruzione, calcola:

```
BP + 00h = 0008h + 00h = 0008h
```

Poi accede in memoria all'indirizzo **DS:0008h** (cioe' a **0CA5h:0008h**), legge gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**; in assenza del codice **3Eh**, la **CPU** avrebbe associato l'offset **0008h** al registro **SS**.

Nell'esempio appena illustrato, il dato **Variabile8** e' stato gestito con un indirizzo **FAR** contenuto in **DS:BP**; cio' e' una diretta conseguenza del fatto che **BP**, non viene associato automaticamente a **DS**.

La situazione e' perfettamente analoga nel caso di un **effective address** del tipo **[base+index+disp]**; supponiamo, ad esempio, di voler accedere a **Variabile8** attraverso **[BX+SI+Disp]**, con **BX=0004h**, **SI=0002h** e **Disp=0002h**. Se **DATASEGM** e' stato caricato in **DS**, possiamo evitare il

segment override scrivendo:
```
mov bl, [bx+si+0002h]
```
Osserviamo che in questa istruzione, figura il registro (puntatore) **BX** come sorgente, e il registro **BL** come destinazione; ovviamente, dopo il trasferimento dati, gli **8** bit meno significativi dell'offset **0004h** contenuto in **BX**, vengono sovrascritti.
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e da un **Disp** a **8** bit; infatti (Capitolo 11), **0002h** e' minore di **127**, per cui l'assembler utilizza un **Disp8=02h**. Nel nostro caso, l'istruzione e' **to register**, per cui **d=1**; grazie a **BL**, l'assembler capisce che gli operandi sono a **8** bit (**w=0**).
Otteniamo quindi:
```
Opcode = 10001010b = 8Ah
```
Il registro destinazione e' **BL**, per cui **reg=011b**; la sorgente e' un **effective address** del tipo **[BX+SI+Disp8]**, per cui **mod=01b**, **r/m=000b**. Otteniamo allora:
```
mod_reg_r/m = 01011000b = 58h
```
Il **Disp8** e':
```
Disp8 = 00000010b = 02h
```
In assenza di diverse indicazioni da parte del programmatore, la **base BX** viene associata automaticamente a **DS**, per cui non e' necessario nessun segment override; l'assembler genera quindi il codice macchina:
```
10001010b 01011000b 00000010b = 8Ah 58h 02h
```
Quando la **CPU** incontra questa istruzione, calcola:
```
BX + SI + 02h = 0004h + 0002h + 02h = 0008h
```
Poi accede in memoria all'indirizzo **DS:0008h** (cioe' a **0CA5h:0008h**), legge gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**.

Nell'esempio appena illustrato, il dato **Variabile8** e' stato gestito con un indirizzo **NEAR** contenuto in **(BX+SI+Disp)**; cio' e' una diretta conseguenza del fatto che la **base BX**, viene associata automaticamente a **DS**.

Se abbiamo caricato **DATASEGM** in **ES**, la precedente istruzione deve essere scritta, necessariamente, come:
```
mov bl, es:[bx+si+0002h]
```
La **base BX** non viene associata automaticamente a **ES**, per cui l'assembler deve generare un codice macchina simile al precedente, ma preceduto dal segment override **26h** relativo allo stesso **ES**; in caso contrario, la **base BX** verrebbe associata automaticamente a **DS**. Si ottiene quindi:
```
00100110b 10001010b 01011000b 00000010b = 26h 8Ah 58h 02h
```
Quando la **CPU** incontra questa istruzione, calcola:
```
BX + SI + 02h = 0004h + 0002h + 02h = 0008h
```
Poi accede in memoria all'indirizzo **ES:0008h** (cioe' a **0CA5h:0008h**), legge gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**; in assenza del codice **26h**, la **CPU** avrebbe associato l'offset **0008h** al registro **DS**.

Nell'esempio appena illustrato, il dato **Variabile8** e' stato gestito con un indirizzo **FAR** contenuto in **ES:(BX+SI+Disp)**; cio' e' una diretta conseguenza del fatto che la **base BX**, non viene associata automaticamente a **ES**.

Supponiamo infine di voler accedere a **Variabile8** attraverso **[BP+DI+Disp]**, con **BP=0004h**, **DI=0002h** e **Disp=0002h**; anche se **DATASEGM** e' stato caricato in **DS**, non possiamo evitare il segment override, e dobbiamo per forza scrivere:
```
mov bl, ds:[bp+di+0002h]
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e da un **Disp8=02h**; nel nostro caso, l'istruzione e' **to register**, per cui **d=1**. Grazie a **BL**, l'assembler capisce che gli operandi sono a **8** bit (**w=0**);

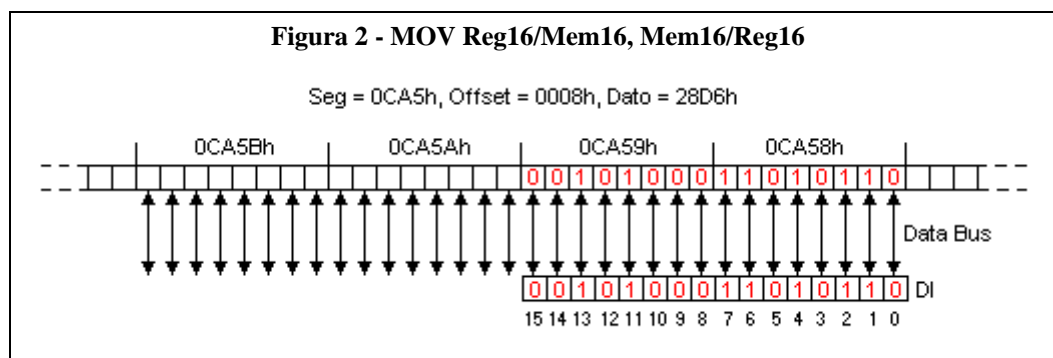
otteniamo quindi:
`Opcode = 10001010b = 8Ah`
Il registro destinazione e' **BL**, per cui **reg=011b**; la sorgente e' un **effective address** del tipo **[BP+DI+Disp8]**, per cui **mod=01b**, **r/m=011b**. Otteniamo allora:
`mod_reg_r/m = 01011011b = 5Bh`
Il **Disp8** e':
`Disp8 = 00000010b = 02h`
La **base BP** non viene associata automaticamente a **DS**, per cui l'assembler deve generare un codice macchina preceduto dal segment override **3Eh** relativo allo stesso **DS**; in caso contrario, la **base BP** verrebbe associata automaticamente a **SS**. L'assembler genera quindi il codice macchina:
`00111110b 10001010b 01011011b 00000010b = 3Eh 8Ah 5Bh 02h`
Quando la **CPU** incontra questa istruzione, calcola:
`BP + DI + 02h = 0004h + 0002h + 02h = 0008h`
Poi accede in memoria all'indirizzo **DS:0008h** (cioe' a **0CA5h:0008h**), legge gli **8** bit **11010110b**, e li trasferisce negli **8** bit del registro **BL**; in assenza del codice **3Eh**, la **CPU** avrebbe associato l'offset **0008h** al registro **SS**.

Nell'esempio appena illustrato, il dato **Variabile8** e' stato gestito con un indirizzo **FAR** contenuto in **DS:(BP+DI+Disp)**; cio' e' una diretta conseguenza del fatto che la **base BP**, non viene associata automaticamente a **DS**.

Appare ovvio il fatto che, se specifichiamo indirizzamenti del tipo:
DS:[SI], **DS:[DI+Disp]**, **DS:[BX+SI+Disp]**, etc, l'assembler non inserisce nessun segment override; infatti, **DI**, **SI** e **BX** vengono associati automaticamente a **DS** (in sostanza, in questi casi l'indicazione esplicita di **DS** e' ridondante).

16.1.2 Il dato si trova in CODESEGM

Supponiamo che all'offset **0008h** del blocco **CODESEGM** del nostro programma, sia presente la seguente definizione:
`Variabile16 dw ? ; dato non inizializzato a 16 bit`
Quando l'assembler incontra questa definizione, crea all'offset **0008h** di **CODESEGM**, una locazione da **16** bit, nella quale non viene inserito nessun valore iniziale.
Supponiamo inoltre che quando inizia l'esecuzione del programma, si abbia:
`CODESEGM = 0CA5h`
Di conseguenza, il dato **Variabile16** viene caricato in memoria, all'indirizzo logico **CODESEGM:0008h**, cioe' **0CA5h:0008h**.
Ad un certo punto del nostro programma, abbiamo la necessita' di effettuare un trasferimento dati dal registro **DI=28D6h** a **Variabile16**; analizziamo allora i vari metodi di indirizzamento che ci permettono di svolgere questa operazione. La Figura 2 illustra in modo schematico, come avviene il trasferimento dati da **Variabile16** a **DI** e viceversa, attraverso un **Data Bus** a **32** bit.

Figura 2 - MOV Reg16/Mem16, Mem16/Reg16

E' importante ricordare che tutte le **CPU** della famiglia **80x86**, seguono la convenzione **little endian** per la disposizione dei dati in memoria; in base a tale convenzione, un dato di tipo **WORD**, **DWORD**, etc, viene disposto in memoria in modo che il suo **BYTE** meno significativo occupi l'indirizzo piu' basso. In Figura 2, ad esempio, osserviamo che il dato a **16** bit **28D6h**, viene disposto in memoria con il **BYTE** meno significativo **D6h** che occupa l'indirizzo fisico **0CA58h**, e il **BYTE** piu' significativo **28h** che occupa l'indirizzo fisico **0CA59h**.
Sempre in base alle convenzioni seguite dalle **CPU** della famiglia **80x86**, l'indirizzo fisico di un dato di tipo **WORD**, **DWORD**, etc, coincide con l'indirizzo fisico del suo **BYTE** meno significativo; nel caso di Figura 2, l'indirizzo fisico del dato **28D6h**, e' **0CA58h**.

La Figura 2 ci permette anche di ricordare che i registri **BX**, **SI**, **DI** e **BP**, possono essere utilizzati, sia come registri puntatori, sia come normalissimi registri generali, destinati a contenere dei generici valori numerici; nel nostro esempio, infatti, stiamo utilizzando **DI** come registro generale.

Nel momento in cui la **CPU** sta eseguendo le istruzioni presenti in **CODESEGM**, si ha, ovviamente, **CS=CODESEGM**; e' necessario ricordare sempre che il compito di gestire **CS** spetta rigorosamente al **SO** e alla **CPU** e non al programmatore.
Nei precedenti capitoli, abbiamo visto che gli assembler come **MASM** e **TASM**, all'inizio di ogni segmento di codice, come **CODESEGM**, richiedono una direttiva del tipo:
```
ASSUME CS: CODESEGM
```
In base a queste premesse, possiamo subito intuire che per accedere al dato **Variabile16**, non possiamo fare a meno del segment override; nel caso piu' semplice, possiamo scrivere l'istruzione:
```
mov Variabile16, di
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e da un **Disp16**; nel nostro caso, l'istruzione e' **from register** (**d=0**), e gli operandi sono a **16** bit (**w=1**). Otteniamo quindi:
```
Opcode = 10001001b = 89h
```
Il registro sorgente e' **DI**, per cui **reg=111b**; la destinazione e' un **Disp16**, per cui **mod=00b**, **r/m=110b**. Otteniamo allora:
```
mod_reg_r/m = 00111110b = 3Eh
```
Il **Disp16** e':
```
Disp16 = 0000000000001000b = 0008h
```
Grazie alla precedente direttiva **ASSUME**, l'assembler tratta **Variabile16** come un **Disp16** riferito a **CS**; non essendo **CS** il registro di segmento predefinito per i dati, si rende necessario il segment override **2Eh**, relativo allo stesso **CS**, altrimenti, la **CPU** assocerebbe automaticamente l'offset **0008h** a **DS**. L'assembler genera quindi il codice macchina:
```
00101110b 10001001b 00111110b 0000000000001000b = 2Eh 89h 3Eh 0008h
```
Quando la **CPU** incontra questa istruzione, legge i **16** bit **0010100011010110b** contenuti in **DI** e li trasferisce nella locazione da **16** bit che si trova in memoria all'indirizzo **CS:0008h** (cioe' **0CA5h:0008h**); in assenza del codice **2Eh**, la **CPU** avrebbe associato l'offset **0008h** a **DS**.

Nell'esempio appena illustrato, il dato **Variabile16** e' stato gestito con un indirizzo di tipo **FAR**; cio' e' una diretta conseguenza del fatto che **CS** non e' il registro di segmento predefinito per i dati.

La situazione non cambia nel momento in cui decidiamo di utilizzare i registri puntatori; indipendentemente quindi dal registro puntatore utilizzato, siamo obbligati a specificare in modo esplicito il segment override **CS**.
Supponiamo, ad esempio, di caricare in **SI** l'offset di **Variabile16**; Vediamo allora quello che succede quando l'assembler incontra l'istruzione:
```
mov cs:[si], di
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m**; nel nostro caso, l'istruzione e' **from register**, per cui

d=0. Grazie alla presenza di **DI**, l'assembler capisce che gli operandi sono a **16** bit, per cui **w=1**; otteniamo quindi:
```
Opcode = 10001001b = 89h
```
Il registro sorgente e' **DI**, per cui **reg=111b**; la destinazione e' un **effective address** del tipo **[SI]**, per cui **mod=00b**, **r/m=100b**. Otteniamo allora:
```
mod_reg_r/m = 00111100b = 3Ch
```
Il registro **SI**, non viene associato automaticamente a **CS**, per cui l'assembler, genera un codice macchina preceduto dal segment override **2Eh** relativo allo stesso **CS**; in caso contrario, **SI** verrebbe associato automaticamente a **DS**. L'assembler genera quindi il codice macchina:
```
00101110b 10001001b 00111100b = 2Eh 89h 3Ch
```
Quando la **CPU** incontra questa istruzione, legge i **16** bit **0010100011010110b** contenuti in **DI** e li trasferisce nella locazione da **16** bit che si trova in memoria all'indirizzo **CS:SI** (cioe' **0CA5h:0008h**); in assenza del codice **2Eh**, la **CPU** avrebbe associato l'offset **0008h** a **DS**.

Nell'esempio appena illustrato, il dato **Variabile16** e' stato gestito con un indirizzo **FAR** contenuto in **CS:SI**; cio' e' una diretta conseguenza del fatto che **SI**, non viene associato automaticamente a **CS**.

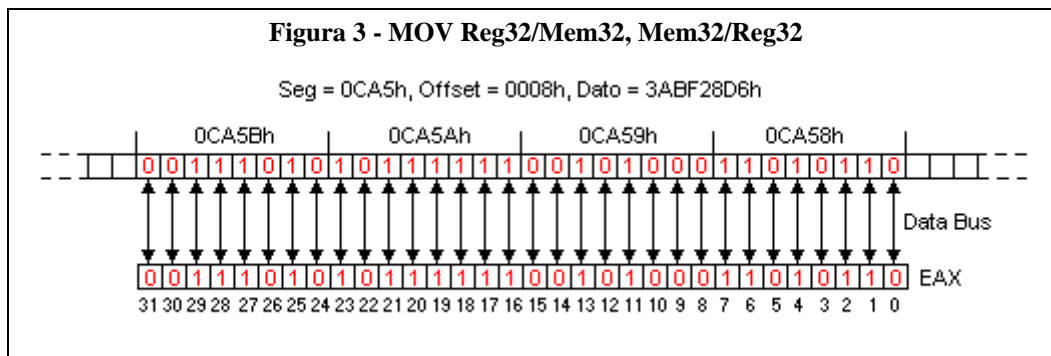
Supponiamo infine di voler accedere a **Variabile16** attraverso **[BP+SI+Disp]**, con **BP=0004h**, **SI=0002h** e **Disp=0002h**; anche in questo caso, non possiamo evitare il segment override, e dobbiamo per forza scrivere:
```
mov cs:[bp+si+0002h], di
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e da un **Disp8=02h**; nel nostro caso, l'istruzione e' **from register**, per cui **d=0**. Grazie alla presenza di **DI**, l'assembler capisce che gli operandi sono a **16** bit, per cui **w=1**; otteniamo quindi:
```
Opcode = 10001001b = 89h
```
Il registro sorgente e' **DI**, per cui **reg=111b**; la destinazione e' un **effective address** del tipo **[BP+SI+Disp8]**, per cui **mod=01b**, **r/m=010b**. Otteniamo allora:
```
mod_reg_r/m = 01111010b = 7Ah
```
Il **Disp8** e':
```
Disp8 = 00000010b = 02h
```
La **base BP**, non viene associata automaticamente a **CS**, per cui l'assembler, genera un codice macchina preceduto dal segment override **2Eh** relativo allo stesso **CS**; in caso contrario, la **base BP** verrebbe associata automaticamente a **SS**. L'assembler genera quindi il codice macchina:
```
00101110b 10001001b 01111010b 00000010b = 2Eh 89h 7Ah 02h
```
Quando la **CPU** incontra questa istruzione, legge i **16** bit **0010100011010110b** contenuti in **DI** e li trasferisce nella locazione da **16** bit che si trova in memoria all'indirizzo **CS:(BP+SI+0002h)** (cioe' **0CA5h:0008h**); in assenza del codice **2Eh**, la **CPU** avrebbe associato l'offset **0008h** a **DS**.

Nell'esempio appena illustrato, il dato **Variabile16** e' stato gestito con un indirizzo **FAR** contenuto in **CS:(BP+SI+0002h)**; cio' e' una diretta conseguenza del fatto che la **base BP**, non viene associata automaticamente a **CS**.

16.1.3 Il dato si trova in STACKSEGM

Supponiamo che all'offset **0008h** del blocco **STACKSEGM** del nostro programma, sia presente la seguente definizione:
```
Variabile32 dd 3ABF28D6h ; 00111010101111110010100011010110b
```
Quando l'assembler incontra questa definizione, crea all'offset **0008h** di **STACKSEGM**, una locazione da **32** bit, nella quale inserisce il valore iniziale **3ABF28D6h**, cioe' **00111010101111110010100011010110b**.
Supponiamo inoltre che quando inizia l'esecuzione del programma, si abbia:

```
STACKSEGM = 0CA5h
```
Di conseguenza, il dato **Variabile32** viene caricato in memoria, all'indirizzo logico **STACKSEGM:0008h**, cioe' **0CA5h:0008h**.
Ad un certo punto del nostro programma, abbiamo la necessita' di effettuare un trasferimento dati da **Variabile32** al registro accumulatore **EAX**; analizziamo allora i vari metodi di indirizzamento che ci permettono di svolgere questa operazione. La Figura 3 illustra in modo schematico, come avviene il trasferimento dati da **Variabile32** a **EAX** e viceversa, attraverso un **Data Bus** a **32** bit.

Figura 3 - MOV Reg32/Mem32, Mem32/Reg32

In base alle convenzioni citate in precedenza, in Figura 3 l'indirizzo fisico del dato **3ABF28D6h** e' **0CA58h**; osserviamo, inoltre, che il dato e' allineato alla **DWORD**, per cui la sua lettura o scrittura, richiede un unico accesso in memoria da parte della **CPU**.

Se il segmento **STACKSEGM** ha l'attributo di combinazione **STACK**, allora il **SO** pone **SS=STACKSEGM**; in caso contrario, il compito di inizializzare **SS** spetta al programmatore, che a sua volta, deve porre **SS=STACKSEGM**
Se vogliamo servirci dell'identificatore **Variabile32**, dobbiamo inserire nel blocco codice una direttiva del tipo:
```
ASSUME SS: STACKSEGM
```
In base a queste premesse, possiamo dire che per accedere nel modo piu' semplice possibile al dato **Variabile32**, possiamo scrivere l'istruzione:
```
mov eax, Variabile32
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 1010000w**, seguito da un **Disp16** (forma compatta, dovuta alla presenza dell'accumulatore); nel nostro caso, gli operandi sono a **32** bit (**full size**), per cui **w=1**. Otteniamo quindi:
```
Opcode = 10100001b = A1h
```
Il **Disp16** e':
```
Disp16 = 0000000000001000b = 0008h
```
Grazie alla precedente direttiva **ASSUME**, l'assembler tratta **Variabile32** come un **Disp16** riferito a **SS**; non essendo **SS** il registro di segmento predefinito per i dati, si rende necessario il segment override **36h**, relativo allo stesso **SS**. Gli operandi sono a **32** bit, per cui e' necessario anche il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
01100110b 00110110b 10100001b 0000000000001000b = 66h 36h A1h 0008h
```
Quando la **CPU** incontra questa istruzione, associa l'offset **0008h** a **SS**, accede in memoria all'indirizzo **SS:0008h** (cioe' a **0CA5h:0008h**), legge i **32** bit **00111010101111110010100011010110b** e li trasferisce nei **32** bit di **EAX**; in assenza del codice **36h**, la **CPU** avrebbe associato l'offset **0008h** a **DS**.

Se omettiamo la precedente direttiva **ASSUME**, siamo costretti a scrivere in modo esplicito:
```
mov eax, ss:Variabile32
```
In tal caso, l'assembler genera lo stesso identico codice macchina precedente.

Nell'esempio appena illustrato, il dato **Variabile32** e' stato gestito con un indirizzo di tipo **FAR**; cio' e' una diretta conseguenza del fatto che **SS** non e' il registro di segmento predefinito per i dati.

Se decidiamo di utilizzare i registri puntatori, possiamo evitare o meno il segment override; infatti, **BX**, **SI** e **DI** vengono associati automaticamente a **DS**, mentre **BP** viene associato automaticamente a **SS**.
Supponiamo, ad esempio, di caricare in **SI** l'offset di **Variabile32**; in tal caso, il segment override e' necessario, e dobbiamo quindi scrivere l'istruzione:
```
mov eax, ss:[si]
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m**; nel nostro caso, l'istruzione e' **to register**, per cui **d=1**. Grazie alla presenza di **EAX**, l'assembler capisce che gli operandi sono a **32** bit, per cui **w=1** (**full size**); otteniamo quindi:
```
Opcode = 10001011b = 8Bh
```
Il registro destinazione e' **EAX**, per cui **reg=000b**; la sorgente e' un **effective address** del tipo **[SI]**, per cui **mod=00b**, **r/m=100b**. Otteniamo allora:
```
mod_reg_r/m = 00000100b = 04h
```
Il registro **SI**, non viene associato automaticamente a **SS**, per cui l'assembler, genera un codice macchina preceduto dal segment override **36h** relativo allo stesso **SS**; in caso contrario, **SI** verrebbe associato automaticamente a **DS**. Gli operandi sono a **32** bit, per cui e' necessario anche il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
01100110b 00110110b 10001011b 00000100b = 66h 36h 8Bh 04h
```
Quando la **CPU** incontra questa istruzione, legge da **SI** l'offset **0008h**, accede in memoria all'indirizzo **SS:0008h** (cioe' a **0CA5h:0008h**), legge i **32** bit **00111010101111110010100011010110b**, e li trasferisce nei **32** bit del registro **EAX**; in assenza del codice **36h**, la **CPU** avrebbe associato l'offset **0008h** al registro **DS**.

Nell'esempio appena illustrato, il dato **Variabile32** e' stato gestito con un indirizzo **FAR** contenuto in **SS:DI**; cio' e' una diretta conseguenza del fatto che **DI**, non viene associato automaticamente a **SS**.

Supponiamo infine di voler accedere a **Variabile32** attraverso **[BP+SI+Disp]**, con **BP=0004h**, **SI=0002h** e **Disp=0002h**; in questo caso, il segment override viene evitato, in quanto la **base BP**, viene associata automaticamente a **SS**. Possiamo quindi scrivere l'istruzione:
```
mov eax, [bp+si+0002h]
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e da un **Disp8=02h**; nel nostro caso, l'istruzione e' **to register**, per cui **d=1**. Grazie alla presenza di **EAX**, l'assembler capisce che gli operandi sono a **32** bit, per cui **w=1** (**full size**); otteniamo quindi:
```
Opcode = 10001011b = 8Bh
```
Il registro destinazione e' **EAX**, per cui **reg=000b**; la sorgente e' un **effective address** del tipo **[BP+SI+Disp8]**, per cui **mod=01b**, **r/m=010b**. Otteniamo allora:
```
mod_reg_r/m = 01000010b = 42h
```
Il **Disp8** e':
```
Disp8 = 00000010b = 02h
```
La **base BP** viene associata automaticamente a **SS**, per cui il segment override non e' necessario; gli operandi sono a **32** bit, per cui e' necessario il prefisso **66h**. L'assembler genera quindi il codice macchina:
```
01100110b 10001011b 01000010b 00000010b = 66h 8Bh 42h 02h
```
Quando la **CPU** incontra questa istruzione, legge da **(BP+SI+02h)** l'offset **0008h**, accede in memoria all'indirizzo **SS:0008h** (cioe' a **0CA5h:0008h**), legge i **32** bit **00111010101111110010100011010110b**, e li trasferisce nei **32** bit del registro **EAX**.

Nell'esempio appena illustrato, il dato **Variabile32** e' stato gestito con un indirizzo **NEAR** contenuto in **(BP+SI+02h)**; cio' e' una diretta conseguenza del fatto che la **base BP**, viene associata automaticamente a **SS**.

Nota importante.
Se si utilizza **MASM**, si sconsiglia vivamente di definire dati statici all'interno di un segmento di stack suddiviso in due o piu' parti che si trovano distribuite in moduli differenti; in questo caso infatti, il linker di **MASM**, nell'unire le varie parti, sovrascrive i dati statici.
In generale, e' opportuno che si eviti del tutto la definizione di dati statici nel segmento di stack; se proprio non si vuole fare a meno di questa possibilita', si consiglia di creare un segmento di stack formato da un unico blocco, che deve trovarsi in un unico file (generalmente, nel file che contiene il programma principale).
Come e' stato spiegato in un precedente capitolo, e' importante che un segmento misto **data+stack**, venga organizzato in modo da garantire l'integrita' delle varie informazioni in esso contenute; in sostanza, i dati statici devono essere disposti nella parte iniziale del segmento (cioe', agli indirizzi piu' bassi), mentre i dati temporanei devono essere disposti nella parte finale del segmento (cioe', agli indirizzi piu' alti).
Tutte le definizioni relative ai dati statici (comprese le definizioni di eventuali buchi di memoria), devono essere rigorosamente seguite dalla inizializzazione; se il valore iniziale non e' importante, si puo' utilizzare uno **0**.

Nota importante.
Dagli esempi appena esposti, risulta evidente che il programmatore **Assembly**, per poter gestire in modo corretto gli indirizzamenti relativi ad operandi di tipo **Mem**, deve attenersi a poche semplicissime regole; tali regole, sono una diretta conseguenza del fatto che la **CPU** tratta, **CS** come **SegReg** predefinito per il codice, **DS** come **SegReg** predefinito per i dati, **SS** come **SegReg** predefinito per lo stack. I registri puntatori, risultano legati a questi tre **SegReg**, dalle associazioni predefinite che abbiamo analizzato in precedenza; ogni volta che vogliamo aggirare tutte queste regole, non dobbiamo fare altro che ricorrere al segment override.

Se vogliamo servirci della libreria **EXELIB** per verificare in pratica gli esempi appena esposti, possiamo ricorrere ad un semplice espediente che ci permette di posizionare i dati all'offset desiderato; in riferimento alla Figura 1, all'inizio del blocco **DATASEGM**, dotato di attributo di allineamento **PARA**, se vogliamo creare il dato **Variabile8** all'offset **0008h**, possiamo scrivere:

```
DATASEGM     SEGMENT  PARA PUBLIC USE16 'DATA'

             db    8 dup (0)    ; buco di memoria da 8 byte
Variabile8   db    0D6h         ; dato all'offset 0008h
```

In riferimento alla Figura 2, all'inizio del blocco **CODESEGM**, dotato di attributo di allineamento **PARA**, se vogliamo creare il dato **Variabile16** all'offset **0008h**, possiamo scrivere prima dell'**entry point**:

```
CODESEGM     SEGMENT  PARA PUBLIC USE16 'CODE'

   assume    cs: CODESEGM      ; associa CODESEGM a CS

             db    8 dup (0)   ; buco di memoria da 8 byte
Variabile16 dw    ?            ; dato all'offset 0008h

start:                         ; entry point
```

In riferimento alla Figura 3, all'inizio del blocco **STACKSEGM**, dotato di attributo di allineamento **PARA**, se vogliamo creare il dato **Variabile32** all'offset **0008h**, possiamo scrivere:

```
STACKSEGM    SEGMENT  PARA PUBLIC USE16 'STACK'

             db    8 dup (0)   ; buco di memoria da 8 byte
Variabile32 dd    3ABF28D6h    ; dato all'offset 0008h
```

Nota importante.
Il **MASM**, esegue l'assemblaggio di un modulo **Assembly**, in due passaggi consecutivi; in questo modo, e' possibile generare il codice macchina di istruzioni che fanno riferimento a identificatori definiti piu' avanti nel programma (**forward references**). Il **TASM** invece, per motivi di efficienza, esegue l'assemblaggio di un modulo in un solo passaggio; nel caso allora di un **forward reference**, il **TASM** genera un messaggio di errore del tipo:
Forward reference needs override
In relazione ai precedenti esempi, questa situazione si verifica quando il blocco **STACKSEGM**, contenente la definizione di **Variabile32**, viene posizionato dopo il blocco **CODESEGM**; in tal caso, una istruzione presente in **CODESEGM**, che fa' riferimento al nome **Variabile32** (definito piu' avanti), genera appunto un messaggio di errore.
Per evitare questo problema, bisogna passare a **TASM** l'opzione **/m**, che permette di specificare il numero di passaggi necessari per l'assemblaggio di un modulo; nel nostro caso, sono sufficienti due passaggi, per cui l'opzione da utilizzare e' **/m2**.

16.1.4 Trasferimento dati da Imm a Reg/Mem

Un aspetto molto interessante da analizzare, riguarda il caso di una istruzione **MOV** (o di una qualsiasi altra istruzione), che comprende un operando sorgente di tipo **Imm**; in particolare, analizziamo il procedimento seguito dall'assembler, per adattare l'operando sorgente **Imm** all'ampiezza in bit dell'operando destinazione.
Ricordiamo che, se **Imm** e' un numero positivo, l'estensione della sua ampiezza, da **m** bit a **n** bit, consiste nell'aggiunta di **n-m** cifre **0** alla sua sinistra; se **Imm** e' un numero negativo, l'estensione della sua ampiezza, da **m** bit a **n** bit, consiste nell'aggiunta di **n-m** cifre **1** alla sua sinistra.

Prima di tutto, inseriamo nel nostro programma la seguente dichiarazione:
```
TEMPERATURA = +25
```
Con questa dichiarazione, stiamo dicendo all'assembler che l'identificatore **TEMPERATURA**, rappresenta un generico valore numerico pari a **+25**; l'importante compito di stabilire l'ampiezza in bit di questo valore, spetta all'assembler.

Nel caso di un operando destinazione di tipo **Reg**, la situazione e' molto semplice; consideriamo, ad esempio, la seguente istruzione:
```
mov dl, TEMPERATURA
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 1011_w_reg**, seguito dal campo **Imm**; grazie alla presenza di **DL**, l'assembler capisce che gli operandi sono a **8** bit, per cui **w=0**. Il registro destinazione e' **DL**, per cui **reg=010b**; otteniamo quindi:
```
Opcode = 10110010b = B2h
```
Siccome gli operandi sono a **8** bit, l'assembler deve esprimere il valore **+25** sotto forma di **Imm8**; otteniamo quindi:
```
Imm8 = 00011001b = 19h
```
L'assembler genera quindi il codice macchina:
```
10110010b 00011001b = B2h 19h
```
Quando la **CPU** incontra questa istruzione, trasferisce nel registro **DL** gli **8** bit del valore immediato **00011001b** (incorporato nel codice macchina dell'istruzione stessa).

Consideriamo l'istruzione:
```
mov dx, TEMPERATURA
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 1011_w_reg**, seguito dal campo **Imm**; grazie alla presenza di **DX**, l'assembler capisce che gli operandi sono a **16** bit, per cui **w=1**. Il registro destinazione e' **DX**, per cui **reg=010b**; otteniamo quindi:
```
Opcode = 10111010b = BAh
```
Siccome gli operandi sono a **16** bit, l'assembler deve esprimere il valore **+25** sotto forma di **Imm16**; otteniamo quindi:
```
Imm16 = 0000000000011001b = 0019h
```
L'assembler genera quindi il codice macchina:
```
10111010b 0000000000011001b = BAh 0019h
```
Quando la **CPU** incontra questa istruzione, trasferisce nel registro **DX** i **16** bit del valore immediato **0000000000011001b** (incorporato nel codice macchina dell'istruzione stessa).

Consideriamo l'istruzione:
```
mov edx, TEMPERATURA
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 1011_w_reg**, seguito dal campo **Imm**; grazie alla presenza di **EDX**, l'assembler capisce che gli operandi sono a **32** bit, per cui **w=1** (**full size**). Il registro destinazione e' **EDX**, per cui **reg=010b**; otteniamo quindi:
```
Opcode = 10111010b = BAh
```
Siccome gli operandi sono a **32** bit, l'assembler deve esprimere il valore **+25** sotto forma di **Imm32**; otteniamo quindi:
```
Imm32 = 00000000000000000000000000011001b = 00000019h
```
Per indicare alla **CPU** che gli operandi sono a **32** bit, e' necessario il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
01100110b 10111010b 00000000000000000000000000011001b = 66h BAh 00000019h
```
Quando la **CPU** incontra questa istruzione, trasferisce nel registro **EDX** i **32** bit del valore immediato **00000000000000000000000000011001b** (incorporato nel codice macchina dell'istruzione stessa).

Cosa succede se dichiariamo **TEMPERATURA** come:
`TEMPERATURA = -25` ?
Con questa dichiarazione, stiamo dicendo all'assembler che l'identificatore **TEMPERATURA**, rappresenta un generico valore numerico con segno, pari a **-25**; questa volta, l'assembler deve

stabilire l'ampiezza in bit di **TEMPERATURA**, rispettandone anche il segno negativo.

Ripetendo gli esempi precedenti, possiamo subito intuire che nel codice macchina, cambia solamente il valore assunto dall'operando **Imm**; partiamo allora dall'istruzione:
```
mov dl, TEMPERATURA
```
Siccome gli operandi sono a **8** bit, l'assembler deve esprimere il valore **-25** sotto forma di **Imm8** in complemento a **2**; otteniamo quindi:
```
Imm8 = 256 - 25 = 231 = 11100111b = E7h
```
L'assembler genera quindi il codice macchina:
```
10110010b 11100111b = B2h E7h
```
Quando la **CPU** incontra questa istruzione, trasferisce nel registro **DL** gli **8** bit del valore immediato **11100111b** (incorporato nel codice macchina dell'istruzione stessa).

Consideriamo l'istruzione:
```
mov dx, TEMPERATURA
```
Siccome gli operandi sono a **16** bit, l'assembler deve esprimere il valore **-25** sotto forma di **Imm16** in complemento a **2**; otteniamo quindi:
```
Imm16 = 65536 - 25 = 65511 = 1111111111100111b = FFE7h
```
L'assembler genera quindi il codice macchina:
```
10111010b 1111111111100111b = BAh FFE7h
```
Quando la **CPU** incontra questa istruzione, trasferisce nel registro **DX** i **16** bit del valore immediato **1111111111100111b** (incorporato nel codice macchina dell'istruzione stessa).

Consideriamo l'istruzione:
```
mov edx, TEMPERATURA
```
Siccome gli operandi sono a **32** bit, l'assembler deve esprimere il valore **-25** sotto forma di **Imm32** in complemento a **2**; otteniamo quindi:
```
Imm32 = 4294967296 - 25 = 4294967271 = 11111111111111111111111111100111b =
FFFFFFE7h
```
Per indicare alla **CPU** che gli operandi sono a **32** bit, e' necessario il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
01100110b 10111010b 11111111111111111111111111100111b = 66h BAh FFFFFFE7h
```
Quando la **CPU** incontra questa istruzione, trasferisce nel registro **EDX** i **32** bit del valore immediato **11111111111111111111111111100111b** (incorporato nel codice macchina dell'istruzione stessa).

Se l'operando destinazione e' di tipo **Mem**, l'aspetto piu' importante da tenere in considerazione, e' dato dal fatto che l'assembler, deve essere in grado di determinare l'ampiezza in bit degli operandi; ovviamente, questa informazione deve essere fornita dal programmatore.
Il procedimento piu' semplice da seguire, consiste nel gestire l'operando di tipo **Mem** attraverso un nome simbolico; in riferimento alla Figura 1, consideriamo l'istruzione:
```
mov Variabile8, TEMPERATURA
```
In questo caso, l'assembler e' in grado di determinare facilmente che il trasferimento dati coinvolge due operandi a **8** bit; infatti, il nome **Variabile8** e' stato utilizzato per definire un dato di tipo **BYTE** (con la direttiva **DB**).

La situazione diventa invece ambigua nel caso dell'istruzione:
```
mov [si], TEMPERATURA
```
In questo caso, l'assembler genera un messaggio di errore per indicare che non puo' determinare l'ampiezza in bit degli operandi; infatti, **TEMPERATURA** e' un generico valore numerico di ampiezza imprecisata, mentre **[SI]** indica il contenuto, di ampiezza imprecisata, di una locazione di memoria che si trova all'indirizzo logico **DS:SI**.
Come gia' sappiamo, per risolvere questo problema possiamo ricorrere agli **address size operators**;

se, ad esempio, vogliamo effettuare un trasferimento dati a **16** bit, possiamo scrivere:

```
mov word ptr [si], TEMPERATURA
```

In questo modo, stiamo dicendo all'assembler che vogliamo copiare il valore numerico **TEMPERATURA** a **16** bit, nella locazione di memoria da **16** bit che si trova all'indirizzo logico **DS:SI**.

Nota importante.
Dalle considerazioni appena esposte, risulta evidente che il programmatore **Assembly** deve prestare grande attenzione ai valori numerici assegnati ad un **Imm**; se, ad esempio, vogliamo che **TEMPERATURA** possa essere gestito anche come numero con segno a **8** bit, dobbiamo assegnare a questo **Imm** un valore compreso tra **-128** e **+127**.
Vediamo, infatti, quello che succede se assegnamo a **TEMPERATURA** il valore **+128**; in tal caso, tutto dipende dall'ampiezza in bit degli operandi. Consideriamo, ad esempio, la seguente istruzione:

```
mov cx, TEMPERATURA
```

L'assembler converte **TEMPERATURA** nella rappresentazione a **16** bit di **+128**, e ottiene **0080h**; per i numeri interi con segno a **16** bit, **0080h** e' ancora la rappresentazione del numero positivo **+128**. Infatti, questo insieme numerico e' compreso tra **-32768** e **+32767**, cioe' tra **8000h** e **7FFFh**.
La situazione pero' cambia se scriviamo:

```
mov ch, TEMPERATURA
```

L'assembler converte **TEMPERATURA** nella rappresentazione a **8** bit di **+128**, e ottiene **80h**; ma per i numeri interi con segno a **8** bit, **80h** non rappresenta piu' il numero positivo **+128**, ma bensi' il numero negativo **-128**. Infatti, questo insieme numerico e' compreso tra **-128** e **+127**, cioe' tra **80h** e **7Fh**.

Molti compilatori ed interpreti dei linguaggi di alto livello, sono in grado di segnalare questo tipo di problema (overflow); la filosofia dell'**Assembly** invece, consiste nel lasciare al programmatore la massima liberta' di scelta su questi aspetti.

Nei precedenti capitoli, abbiamo visto che gli operatori **SEG** e **OFFSET** producono un risultato di tipo **Imm16**, per cui possono comparire solo nell'operando sorgente di una istruzione; in sostanza, istruzioni del tipo:

```
mov ax, seg Variabile8
```

oppure:

```
mov Variabile16, offset start
```

non sono altro che trasferimenti di dati da **Imm16** a **Reg16/Mem16**.

16.1.5 Indirizzamenti a 32 bit

Come e' stato spiegato in precedenza, quando si programma in modalita' reale e' opportuno che si evitino gli indirizzamenti con componente **Offset** a **32** bit; nel seguito, ci limitiamo a presentare un semplice esempio che serve anche ad evidenziare un bug del **MASM** (gia' illustrato in un precedente capitolo).
Consideriamo un programma in formato **EXE**, dotato di, un segmento di dati chiamato **DATASEGM**, un segmento di codice chiamato **CODESEGM** e un segmento di stack chiamato **STACKSEGM**; all'offset **0000h** di **DATASEGM**, inseriamo la seguente definizione:

```
VarData32 dd 3C2AB1C8h
```

All'offset **0000h** di **STACKSEGM**, inseriamo la seguente definizione:

```
VarStack32 dd 1CF8442Dh
```

Servendoci della libreria **EXELIB**, nell'ipotesi che **DS=DATASEGM** e **SS=STACKSEGM**, possiamo scrivere:

```
; La base ECX viene associata automaticamente a DS, per cui
; non e' necessario il segment override DS:[ECX+EBP]

mov      ecx, 0                ; ecx = 0
mov      cx, offset VarData32  ; ds:ecx punta a VarData32
mov      ebp, 0                ; ebp = 0
mov      eax, [ecx+ebp]        ; eax = ds:[ecx+ebp] = 3C2AB1C8h
mov      dx, 0400h             ; riga 4, colonna 0
call     writeHex32            ; visualizza 3C2AB1C8h

; La base EBP viene associata automaticamente a SS, per cui
; non e' necessario il segment override SS:[EBP+ECX]

mov      ebp, 0                ; ebp = 0
mov      bp, offset VarStack32 ; ss:ebp punta a VarStack32
mov      ecx, 0                ; ecx = 0
mov      eax, [ebp+ecx]        ; eax = ss:[ebp+ecx] = 1CF8442Dh
mov      dx, 0500h             ; riga 5, colonna 0
call     writeHex32            ; visualizza 1CF8442Dh
```

Utilizzando **TASM**, questo esempio funziona perfettamente; si tenga presente che il linker di **TASM**, in presenza di indirizzi a **32** bit, richiede l'opzione **/3**.
Analizziamo ora il codice macchina generato da **TASM**, per l'istruzione:
`mov eax, [ecx+ebp]`
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e dal **S.I.B.** byte; nel nostro caso, l'istruzione e' **to register**, per cui **d=1**. Grazie a **EAX**, l'assembler capisce che gli operandi sono a **32** bit, per cui **w=1** (**full size**); otteniamo quindi:
`Opcode = 10001011b = 8Bh`
Il registro destinazione e' **EAX**, per cui **reg=000b**; la sorgente e' un **effective address** del tipo:
`[ECX+(indice scalato)]`
per cui **mod=00b**, **r/m=100b**. Otteniamo allora:
`mod_reg_r/m = 00000100b = 04h`
Il fattore di scala e' **1** (nessuna moltiplicazione), il registro base e' **ECX**, mentre il registro indice e' **EBP**; otteniamo allora, **scale=00b**, **index=101b**, **base=001b**. Per il campo **S.I.B.** (**scale_index_base**) si ha quindi:
`S.I.B. = 00101001b = 29h`
Gli operandi sono a **32** bit, per cui l'assembler inserisce il prefisso **66h**; gli offset sono a **32** bit, per cui l'assembler inserisce il prefisso **67h**.
In assenza di diverse indicazioni da parte del programmatore, la **base ECX** viene associata automaticamente a **DS**, per cui non e' necessario nessun segment override; l'assembler genera quindi il codice macchina:
`01100110b 01100111b 10001011b 00000100b 00101001b = 66h 67h 8Bh 04h 29h`
Quando la **CPU** incontra questa istruzione, accede in memoria all'indirizzo **DS:(ECX+EBP)**, legge i **32** bit **3C2AB1C8h**, e li trasferisce nei **32** bit del registro **EAX**.

Analizziamo il codice macchina generato da **TASM**, per l'istruzione:
`mov eax, [ebp+ecx]`
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dall'**Opcode 100010dw**, seguito dal campo **mod_reg_r/m** e dal **S.I.B.** byte; nel nostro caso, l'istruzione e' **to register**, per cui **d=1**. Grazie a **EAX**, l'assembler capisce che gli operandi sono a **32** bit, per cui **w=1** (**full size**); otteniamo quindi:
`Opcode = 10001011b = 8Bh`

Il registro destinazione e' **EAX**, per cui **reg=000b**; la sorgente e' un **effective address** del tipo:
```
[EBP+(indice scalato)+Disp8]
```
per cui **mod=01b**, **r/m=100b**. Infatti (Capitolo 11), in presenza del registro **base EBP**, viene aggiunto un **Disp8=00h**; otteniamo allora:
```
mod_reg_r/m = 01000100b = 44h
```
Il fattore di scala e' **1** (nessuna moltiplicazione), il registro base e' **EBP**, mentre il registro indice e' **ECX**; otteniamo allora, **scale=00b**, **index=001b**, **base=101b**. Per il campo **S.I.B.** (**scale_index_base**) si ha quindi:
```
S.I.B. = 00001101b = 0Dh
```
Il **Disp8** e':
```
Disp8 = 00000000b = 00h
```
Gli operandi sono a **32** bit, per cui l'assembler inserisce il prefisso **66h**; gli offset sono a **32** bit, per cui l'assembler inserisce il prefisso **67h**.
In assenza di diverse indicazioni da parte del programmatore, la **base EBP** viene associata automaticamente a **SS**, per cui non e' necessario nessun segment override; l'assembler genera quindi il codice macchina:
```
01100110b 01100111b 10001011b 01000100b 00001101b 00000000b = 66h 67h 8Bh 44h
0Dh 00h
```
Quando la **CPU** incontra questa istruzione, accede in memoria all'indirizzo **SS:(EBP+ECX+00h)**, legge i **32** bit **1CF8442Dh**, e li trasferisce nei **32** bit del registro **EAX**.

Se proviamo ad utilizzare **MASM**, ci accorgiamo che l'istruzione:
```
mov eax, [ecx+ebp]
```
visualizza **VarStack32**; analogamente, l'istruzione:
```
mov eax, [ebp+ecx]
```
visualizza **VarData32**!
Consultando il **listing file**, scopriamo che **MASM**, per la prima istruzione ha generato il codice macchina:
```
66h 67h 8Bh 44h 0Dh 00h
```
mentre per la seconda istruzione ha generato il codice macchina:
```
66h 67h 8Bh 04h 29h
```
Come si puo' notare, questi due codici macchina appaiono invertiti rispetto a quelli generati dal **Borland TASM**!
E' stato gia' spiegato che questo bug di **MASM**, si verifica solo in modalita' reale, e in assenza di un fattore di scala esplicito; abbiamo anche visto che per evitare questo bug, possiamo servirci del segment override, scrivendo:
```
mov eax, ds:[ecx+ebp]
```
e:
```
mov eax, ss:[ebp+ecx]
```
Le considerazioni appena esposte, inducono a maggior ragione, ad evitare l'uso degli indirizzamenti a **32** bit in modalita' reale.

16.1.6 Effetti provocati da MOV sugli operandi e sui flags

L'esecuzione dell'istruzione **MOV** tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST**; infatti, il vecchio contenuto di **DEST** viene sovrascritto dal contenuto di **SRC**. Il contenuto dell'operando **SRC** rimane inalterato; bisogna pero' prestare particolare attenzione ad istruzioni del tipo:

```
mov si, [si]
```

oppure:

```
mov bh, ss:[bx+di+09h]
```

Per capire bene questa situazione, consideriamo l'esempio di Figura 2; carichiamo l'offset (**0008h**) di **Variabile16** in **SI**, e scriviamo:

```
mov si, cs:[si]
```

In questo esempio, l'operando **DEST** e' **SI**, mentre l'operando **SRC** e' la locazione di memoria che si trova all'indirizzo logico **0CA5h:0008h**, e che contiene il valore **28D6h**; in seguito al trasferimento dati, otteniamo **SI=28D6h**. Il valore **28D6h** che si trova in memoria all'indirizzo **0CA5h:0008h**, viene ovviamente preservato; in relazione, invece, al registro **SI**, succede che:

***** prima del trasferimento dati, **SI=0008h**;
***** dopo il trasferimento dati, **SI=28D6h**.

In sostanza, dopo il trasferimento dati, la coppia **CS:SI** contiene l'indirizzo logico **0CA5h:28D6h**; questa coppia quindi, non punta piu' alla locazione di memoria in cui si trova **Variabile16**.

Il fatto che sia permesso un trasferimento dati da **Reg** a **Reg**, ci permette di scrivere anche istruzioni del tipo:

```
mov dx, dx
```

Come si puo' facilmente constatare, l'esecuzione di una tale istruzione non provoca nessuna modifica del contenuto di **DX**; proprio per questo motivo, l'assembler utilizza spesso questo tipo di istruzioni (in alternativa a **NOP**), per inserire eventuali buchi di memoria all'interno di un segmento di codice. In tal caso, lo scopo della precedente istruzione e' solo quello di occupare **2** byte di memoria (cioe', i due byte **8Bh**, **D2h** del relativo codice macchina).

L'esecuzione dell'istruzione **MOV**, non modifica nessuno dei campi presenti nel **Flags Register**; per chi programma in **Assembly**, e' fondamentale tenere sempre conto di questo aspetto. Ricordiamo, infatti, che in **Assembly** (e in qualunque altro linguaggio di programmazione), il comportamento dei programmi dipende proprio dal contenuto del **Flags Register**; pertanto, si consiglia vivamente di consultare sempre le tabelle, per tenere conto degli effetti prodotti su questo registro dall'esecuzione di una qualsiasi istruzione.

16.2 Le istruzioni MOVZX e MOVSX

Molto spesso, quando si scrive un programma, si presenta la necessita' di modificare l'ampiezza in bit di un numero intero; come si puo' facilmente immaginare, si tratta di un aspetto molto delicato, che deve essere gestito con la massima cautela.
La Figura 4 mostra un programma di esempio scritto in **linguaggio C**.

Figura 4 - Assegnamenti in C

```
#include < stdio.h >

signed char          var8 = -128;   /* intero con segno a 8 bit */
signed short int     var16 = -129;  /* intero con segno a 16 bit */
signed short int     num1;          /* intero con segno a 16 bit */
unsigned short int   num2 = 40000U; /* intero senza segno a 16 bit */

int main(void)
{
   var8 = var16;                    /* var8 = +127 */
   printf("var8 = %d\n", var8);     /* visualizza var8 */
   printf("num2 = %u\n", num2);     /* visualizza num2 */
   num1 = num2;                     /* num1 = -25536 */
   printf("num1 = %d\n", num1);     /* visualizza num1 */

   return 0;
}
```

Come si puo' notare, il **C** permette di effettuare assegnamenti tra variabili di tipo differente; possiamo scrivere, ad esempio:
`var16 = var8`
In questo caso, il contenuto (**-128**) di **var8** viene copiato in **var16**; la conversione non presenta nessun problema, in quanto stiamo passando dagli interi con segno a **8** bit, agli interi con segno a **16** bit. Il contenuto di **var8**, si trova memorizzato nella forma:
`var8 = 256 - 128 = 128 = 10000000b`
Il compilatore **C** effettua l'estensione del bit di segno, e ottiene:
`var16 = 1111111110000000b`
In questo caso, il passaggio da **8** a **16** bit e' stato ottenuto aggiungendo otto **1** alla sinistra di **10000000b**; il risultato che ne scaturisce, e' ancora la rappresentazione di **-128** a **16** bit in complemento a **2**. Infatti:
`-128 = 65536 - 128 = 65408 = 1111111110000000b`

Il discorso pero' cambia se scriviamo:
`var8 = var16`
In questo caso, il contenuto (**-129**) di **var16** viene copiato in **var8**; la conversione comporta un troncamento di cifre significative, in quanto stiamo passando dagli interi con segno a **16** bit, agli interi con segno a **8** bit. Il contenuto di **var16**, si trova memorizzato nella forma:
`var16 = 65536 - 129 = 65407 = 1111111101111111b`
Il compilatore **C** effettua il troncamento degli **8** bit piu' significativi di questo valore, e ottiene:
`var8 = 01111111b`
Nell'insieme dei numeri interi con segno a **8** bit in complemento a due, il valore binario **01111111b** rappresenta il numero positivo **+127**; a causa quindi del troncamento di cifre significative, siamo passati da **-129** a **+127**.

Proviamo ora a scrivere:
`num1 = num2`
In questo caso, stiamo assegnando il valore intero senza segno **40000**, al dato **num2** che e' stato dichiarato come intero con segno; l'ampiezza di entrambe le variabili e' di **16** bit, ma il risultato che

si ottiene, e' sbagliato. Infatti, se proviamo a visualizzare il contenuto di **num1**, otteniamo il numero negativo **-25536**!
Il perche' di questo errore e' abbastanza evidente; per i numeri interi con segno a **16** bit in complemento a **2**, il valore **40000** non e' altro che la rappresentazione di **-25536**. Infatti:

```
-25536 = 65536 - 25536 = 40000 = 1001110001000000b
```

Gli esempi appena presentati, dimostrano la pericolosita' delle conversioni tra dati di tipo differente; il **C** e' un linguaggio destinato agli esperti, per cui assume che il programmatore sappia quello che sta facendo. Anche il linguaggio **Assembly** si basa su questa stessa filosofia; i linguaggi destinati, invece, ai principianti, proibiscono le conversioni tra dati di tipo differente.

Dalle considerazioni appena esposte, risulta evidente che le uniche conversioni sicure, sono quelle che coinvolgono dati dello stesso tipo, e che comportano un aumento della ampiezza in bit del valore da convertire; in caso contrario, e' necessario accertarsi che la conversione non provochi un cambiamento di segno, o una perdita di cifre significative.
Tutte le **CPU** della famiglia **80x86**, forniscono una serie di istruzioni che permettono di estendere in modo sicuro, l'ampiezza in bit di un numero intero con segno; queste istruzioni vengono esaminate nel prossimo capitolo, in quanto fanno parte della categoria delle istruzioni aritmetiche.
In questo capitolo, invece, vengono prese in considerazione due nuove istruzioni, disponibili solo per le **CPU 80386** e superiori, che permettono di eseguire trasferimenti di dati verso un operando **DEST** avente una ampiezza in bit maggiore dell'operando **SRC**; queste due istruzioni, sono rappresentate dai mnemonici **MOVZX** (per gli interi senza segno) e **MOVSX** (per gli interi con segno).

16.2.1 L'istruzione MOVZX

Con il mnemonico **MOVZX** si indica l'istruzione **MOVe data with Zero eXtension** (trasferimento dati con estensione di zeri); quando la **CPU** incontra questa istruzione, legge il contenuto dell'operando **SRC**, ne estende l'ampiezza verso sinistra con degli zeri, e copia il risultato nell'operando **DEST**. Il contenuto di **SRC** quindi, viene trattato sempre come un numero intero senza segno, anche se il suo bit piu' significativo vale **1**; il risultato salvato in **DEST**, e' la rappresentazione ad ampiezza maggiore, del numero intero senza segno contenuto in **SRC**.
L'operando **DEST** deve avere, necessariamente, una ampiezza in bit maggiore di quella dell'operando **SRC**; le uniche combinazioni lecite tra **SRC** e **DEST**, sono le seguenti:

```
MOVZX    Reg16, Reg8
MOVZX    Reg16, Mem8
MOVZX    Reg32, Reg8
MOVZX    Reg32, Mem8
MOVZX    Reg32, Reg16
MOVZX    Reg32, Mem16
```

Come si puo' notare, l'operando **DEST** deve essere, esclusivamente, di tipo **Reg**; inoltre, e' proibito l'uso di operandi di tipo **SegReg** o **Imm**.

In generale, le nuove istruzioni fornite dalle **CPU 80386** e superiori, hanno un campo **Opcode** formato da **2** byte; infatti, il campo **Opcode** da **1** byte utilizzato con le precedenti **CPU**, non e' sufficiente per contenere l'intero set di istruzioni delle **CPU 80386** e superiori.
Nel caso dell'istruzione **MOVZX**, l'**Opcode** e' formato dai **2** byte **00001111** e **1011011w**; **w=0** indica che l'operando **SRC** e' a **8** bit, mentre **w=1** indica che l'operando **SRC** e' a **16** bit.

In riferimento all'esempio di Figura 1, con **Variabile8=76h** (**118d**) e con una direttiva **ASSUME**

che associa **DATASEGM** a **DS**, possiamo scrivere l'istruzione:
```
movzx bx, Variabile8
```
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dal campo **Opcode**, seguito dal campo **mod_reg_r/m** e da un **Disp16**; nel nostro caso, l'operando **SRC** e' a **8** bit, per cui **w=0**; otteniamo quindi:
```
Opcode = 00001111b 10110110b = 0Fh B6h
```
Il registro destinazione e' **BX**, per cui **reg=011b**; la sorgente e' un **Disp16**, per cui **mod=00b**, **r/m=110b**. Otteniamo allora:
```
mod_reg_r/m = 00011110b = 1Eh
```
Il **Disp16** e':
```
Disp16 = 0000000000001000b = 0008h
```
L'operando **DEST** e' a **16** bit, per cui non e' necessario il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
00001111b 10110110b 00011110b 0000000000001000b = 0Fh B6h 1Eh 0008h
```
Quando la **CPU** incontra questa istruzione:

1) accede in memoria all'indirizzo logico **DS:0008h**;
2) legge il valore a **8** bit **01110110b**;
3) estende questo valore a **16** bit con degli **0** a sinistra, ottenendo **0000000001110110b**;
4) copia **0000000001110110b** nei **16** bit di **BX**.

Alla fine, si ottiene in **BX** la rappresentazione a **16** bit del numero intero senza segno **118d**; infatti:
```
118d = 0000000001110110b
```

Se **DS:(BP+SI+0002h)** punta a **Variabile8=D6h** (**214d**), possiamo scrivere l'istruzione:
```
movzx ebx, byte ptr ds:[bp+si+0002h]
```
Il segment override e' necessario, in modo che la **base BP** venga associata a **DS**; l'operatore **BYTE PTR** e' necessario, per poter specificare che l'operando **SRC** e' a **8** bit.
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dal campo **Opcode**, seguito dal campo **mod_reg_r/m** e da un **Disp8**; nel nostro caso, l'operando **SRC** e' a **8** bit, per cui **w=0**; otteniamo quindi:
```
Opcode = 00001111b 10110110b = 0Fh B6h
```
Il registro destinazione e' **EBX**, per cui **reg=011b**; la sorgente e' un **effective address** del tipo **[BP+SI+Disp8]**, per cui **mod=01b**, **r/m=010b**. Otteniamo allora:
```
mod_reg_r/m = 01011010b = 5Ah
```
Il **Disp8** e':
```
Disp8 = 00000010b = 02h
```
La **base BP** non viene associata automaticamente a **DS**, per cui e' necessario il segment override **3Eh** relativo allo stesso **DS**; l'operando **DEST** e' a **32** bit, per cui e' necessario il prefisso **66h**.
L'assembler genera quindi il codice macchina:
```
01100110b 00111110b 00001111b 10110110b 01011010b 00000010b = 66h 3Eh 0Fh B6h
5Ah 02h
```
Quando la **CPU** incontra questa istruzione:

1) accede in memoria all'indirizzo logico **DS:(BP+SI+02h)**;
2) legge il valore a **8** bit **11010110b**;
3) estende questo valore a **32** bit con degli **0** a sinistra, ottenendo **00000000000000000000000011010110b**;
4) copia **00000000000000000000000011010110b** nei **32** bit di **EBX**.

Alla fine, si ottiene in **EBX** la rappresentazione a **32** bit del numero intero senza segno **214d**; infatti:
```
214d = 00000000000000000000000011010110b
```

16.2.2 L'istruzione MOVSX

Con il mnemonico **MOVSX** si indica l'istruzione **MOVe data with Sign eXtension** (trasferimento dati con estensione del bit di segno); quando la **CPU** incontra questa istruzione, legge il contenuto dell'operando **SRC**, ne estende l'ampiezza verso sinistra con il bit di segno di **SRC**, e copia il risultato nell'operando **DEST**. Il contenuto di **SRC** quindi, viene trattato sempre come un numero intero con segno; il risultato salvato in **DEST**, e' la rappresentazione ad ampiezza maggiore, del numero intero con segno contenuto in **SRC**.
L'operando **DEST** deve avere, necessariamente, una ampiezza in bit maggiore di quella dell'operando **SRC**; le uniche combinazioni lecite tra **SRC** e **DEST**, sono le seguenti:

```
MOVSX    Reg16, Reg8
MOVSX    Reg16, Mem8
MOVSX    Reg32, Reg8
MOVSX    Reg32, Mem8
MOVSX    Reg32, Reg16
MOVSX    Reg32, Mem16
```

In relazione alle caratteristiche degli operandi **SRC** e **DEST**, valgono le stesse considerazioni gia' svolte per **MOVZX**.

Per l'istruzione **MOVSX**, l'**Opcode** e' formato dai **2** byte **00001111** e **1011111w**; **w=0** indica che l'operando **SRC** e' a **8** bit, mentre **w=1** indica che l'operando **SRC** e' a **16** bit.

In riferimento all'esempio di Figura 2, con **Variabile16=28D6h** (**+10454d**) e con una direttiva **ASSUME** che associa **CODESEGM** a **CS**, possiamo scrivere l'istruzione:

```
movsx ebx, Variabile16
```

Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dal campo **Opcode**, seguito dal campo **mod_reg_r/m** e da un **Disp16**; nel nostro caso, l'operando **SRC** e' a **16** bit, per cui **w=1**; otteniamo quindi:

```
Opcode = 00001111b 10111111b = 0Fh BFh
```

Il registro destinazione e' **EBX**, per cui **reg=011b**; la sorgente e' un **Disp16**, per cui **mod=00b**, **r/m=110b**. Otteniamo allora:

```
mod_reg_r/m = 00011110b = 1Eh
```

Il **Disp16** e':

```
Disp16 = 0000000000001000b = 0008h
```

Il **Disp16** non viene associato automaticamente a **CS**, per cui e' necessario il segment override **2Eh** relativo allo stesso **CS**; l'operando **DEST** e' a **32** bit, per cui e' necessario il prefisso **66h**.
L'assembler genera quindi il codice macchina:

```
01100110b 00101110b 00001111b 10111111b 00011110b 0000000000001000b = 66h 2Eh
0Fh BFh 1Eh 0008h
```

Quando la **CPU** incontra questa istruzione:

1) accede in memoria all'indirizzo logico **CS:0008h**;
2) legge il valore a **16** bit **0010100011010110b**;
3) estende questo valore a **32** bit con degli **0** a sinistra (bit di segno di **SRC**), ottenendo **00000000000000000010100011010110b**;
4) copia **00000000000000000010100011010110b** nei **32** bit di **EBX**.

Alla fine, si ottiene in **EBX** la rappresentazione a **32** bit in complemento a **2** del numero intero con segno **+10454d**; infatti:

```
+10454d = 00000000000000000010100011010110b
```

Se **CS:(BX+DI+0002h)** punta a **Variabile16=88D6h** (**-30506**), possiamo scrivere l'istruzione:
```
movsx ebx, word ptr cs:[bx+di+0002h]
```
Il segment override e' necessario, in modo che la **base BX** venga associata a **CS**; l'operatore **WORD PTR** e' necessario, per poter specificare che l'operando **SRC** e' a **16** bit.
Dalle tabelle, si ricava che il codice macchina di questa istruzione, e' formato dal campo **Opcode**, seguito dal campo **mod_reg_r/m** e da un **Disp8**; nel nostro caso, l'operando **SRC** e' a **16** bit, per cui **w=1**; otteniamo quindi:
```
Opcode = 00001111b 10111111b = 0Fh BFh
```
Il registro destinazione e' **EBX**, per cui **reg=011b**; la sorgente e' un **effective address** del tipo **[BX+DI+Disp8]**, per cui **mod=01b**, **r/m=001b**. Otteniamo allora:
```
mod_reg_r/m = 01011001b = 59h
```
Il **Disp8** e':
```
Disp8 = 00000010b = 02h
```
La **base BX** non viene associata automaticamente a **CS**, per cui e' necessario il segment override **2Eh** relativo allo stesso **CS**; l'operando **DEST** e' a **32** bit, per cui e' necessario il prefisso **66h**.
L'assembler genera quindi il codice macchina:
```
01100110b 00101110b 00001111b 10111111b 01011001b 00000010b = 66h 2Eh 0Fh BFh
59h 02h
```
Quando la **CPU** incontra questa istruzione:

1) accede in memoria all'indirizzo logico **CS:(BX+DI+02h)**;
2) legge il valore a **16** bit **1000100011010110b**;
3) estende questo valore a **32** bit con degli **1** a sinistra (bit di segno di **SRC**), ottenendo **11111111111111111000100011010110b**;
4) copia **11111111111111111000100011010110b** nei **32** bit di **EBX**.

Alla fine, si ottiene in **EBX** la rappresentazione a **32** bit in complemento a **2** del numero intero con segno **-30506**; infatti:
```
-30506 = 4294967296 - 30506 = 4294936790 = 11111111111111111000100011010110b
```

16.2.3 Inserimento diretto di codici macchina in un programma Assembly

Ogni volta che compare sul mercato una nuova **CPU** della famiglia **80x86**, viene ulteriormente arricchito il set di istruzioni disponibile con le **CPU** precedenti; abbiamo appena visto che con la **80386**, vengono rese disponibili anche le due nuove istruzioni **MOVZX** e **MOVSX**.
Puo' capitare allora di avere a disposizione un assembler che non supporta le nuove istruzioni appartenenti al set della **CPU** installata sul proprio computer; in un caso del genere, e' possibile risolvere il problema senza la necessita' di procurarsi un assembler piu' aggiornato.
La tecnica da utilizzare, consiste nell'inserire direttamente nei propri programmi, il codice macchina delle istruzioni non supportate dall'assembler; come esempio pratico, supponiamo di voler scrivere le seguenti istruzioni:

```
mov      bx, +27580      ; bx = +27580
movsx    ecx, bx         ; ecx = +27580
```

L'assembler in nostro possesso, non supporta pero' l'istruzione **MOVSX**; servendoci allora della documentazione ufficiale della nostra **CPU**, possiamo rilevare che l'istruzione dell'esempio, ha un codice macchina formato dal campo **Opcode** e dal campo **mod_reg_r/m**. L'operando **SRC** e' a **16** bit, per cui **w=1**; otteniamo allora:
```
Opcode = 00001111b 10111111b = 0Fh BFh
```
Il registro destinazione e' **ECX**, per cui **reg=001b**; il registro sorgente e' **BX**, per cui **mod=11b**, **r/m=011b**. Otteniamo allora:
```
mod_reg_r/m = 11001011b = CBh
```

L'operando **DEST** e' a **32** bit, per cui e' necessario il prefisso **66h**; in definitiva, il codice macchina che si ottiene e':
`01100110b 00001111b 10111111b 11001011b = 66h 0Fh BFh CBh`
Dopo aver ricavato queste informazioni, possiamo riscrivere le istruzioni del precedente esempio, in questo modo:

```
mov     bx, +27580              ; bx = +27580
db      66h, 0Fh, 0BFh, 0CBh    ; ecx = +27580
```

Quando l'assembler incontra la direttiva **DB**, crea in quel preciso punto del blocco codice, una sequenza di **4** locazioni consecutive e contigue, ciascuna delle quali occupa **1** byte; in queste **4** locazioni, l'assembler inserisce i valori **66h**, **0Fh**, **BFh**, **CBh**. Quando la **CPU** incontra questo codice macchina, copia in **ECX** il numero intero con segno contenuto in **BX**; naturalmente, affinche' questo esempio possa funzionare, e' necessario disporre almeno di una **CPU 80386**.

Quando si applica questa tecnica, bisogna prestare particolare attenzione ad eventuali operandi di tipo **Disp**; infatti, abbiamo visto che l'assembler marca come **relocatable** questo tipo di operandi, in modo che il linker, se necessario, possa procedere alla loro rilocazione.
Consideriamo un precedente esempio, relativo all'istruzione:
`movsx ebx, Variabile16`
Abbiamo visto che il codice macchina che l'assembler genera per questa istruzione, e':
`66h 2Eh 0Fh BFh 1Eh 0008h`
Anche se sappiamo che **Variabile16** si trova all'offset **0008h** del blocco **CODESEGM**, non possiamo utilizzare questo valore esplicito (che verrebbe trattato dall'assembler come un semplice **Imm16**); dobbiamo fare in modo che sia l'assembler ad inserire l'offset di **Variabile16**, marcandolo anche come **relocatable**. Per ottenere questo risultato, all'interno del blocco **CODESEGM** possiamo inserire il seguente codice macchina:

```
db      66h, 2Eh, 0Fh, 0BFh, 1Eh
dw      offset Variabile16
```

In questo modo, si ottiene il codice macchina completo, che comprende anche l'offset **0008h**, marcato come **relocatable**; infatti, in presenza della direttiva:
`dw offset Variabile16`
l'assembler crea una locazione di memoria da **2** byte, e carica in questa locazione l'offset di **Variabile16**, marcato come **relocatable**.

16.2.4 Effetti provocati da MOVZX e MOVSX sugli operandi e sui flags

L'esecuzione delle istruzioni **MOVZX** e **MOVSX**, tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST**; infatti, il vecchio contenuto di **DEST** viene sovrascritto dal contenuto di **SRC**. Il contenuto dell'operando **SRC** rimane inalterato; come al solito, bisogna prestare particolare attenzione a quelle istruzioni che prevedono lo stesso registro, sia come **SRC**, sia come **DEST**.

Supponendo di avere **CX=-12000** (**1101000100100000b**), consideriamo l'istruzione:
`movsx ecx, cx`
In questo caso, l'operando **DEST** e' **ECX**, mentre l'operando **SRC** e' **CX**; subito dopo l'esecuzione di questa istruzione, otteniamo:
`ECX = 11111111111111111101000100100000b`
e, di conseguenza:
`CX = 1101000100100000b`
Il contenuto di **CX** rimane chiaramente inalterato; infatti, l'istruzione dell'esempio modifica solo i **16** bit piu' significativi di **ECX**.

Supponendo di avere **BX=0008h** e **CS:[BX]=18CBh**, consideriamo la seguente istruzione:

```
movzx ebx, word ptr cs:[bx]
```

In questo caso, l'operando **DEST** e' **EBX**, mentre l'operando **SRC** e' la locazione di memoria puntata da **CS:BX**, e contenente il valore **18CBh**.
L'esecuzione di questa istruzione, preserva il contenuto **18CBh** della locazione di memoria che si trova all'indirizzo **CS:0008h**; per quanto riguarda, invece, l'operando **DEST**, otteniamo **EBX=000018CBh**. Ne consegue che la coppia **CS:BX**, non punta piu' a **CS:0008h**, ma bensi' a **CS:18CBh**!

L'esecuzione delle istruzioni **MOVZX** e **MOVSX**, non modifica nessuno dei campi presenti nel **Flags Register**.

16.3 Le istruzioni PUSH e POP

Con le **CPU** della famiglia **80x86**, il metodo predefinito per la gestione dello stack consiste nell'uso delle istruzioni **PUSH** e **POP**; queste due istruzioni, presuppongono che nel momento in cui inizia la fase di esecuzione di un programma, la coppia **SS:SP** stia puntando alla cima dello stack (**TOS**).
Abbiamo visto che se definiamo un segmento di programma con attributo di combinazione **STACK**, allora la coppia **SS:SP** viene inizializzata automaticamente dal **SO**; in caso contrario, il compito di inizializzare la coppia **SS:SP** spetta a noi.
E' importante ricordare che quando si lavora in modalita' reale con le **CPU** a **32** bit, le componenti **Offset** degli indirizzi logici, non devono superare il valore massimo **0000FFFFh**; quando una **CPU 80386** o superiore viene inizializzata in modalita' reale, i **16** bit piu' significativi del registro **ESP** vengono posti automaticamente a **0000h**.

In generale, l'ampiezza in bit degli operandi di **PUSH** e **POP**, rispecchia fedelmente l'architettura della **CPU**; in questo modo, si semplifica notevolmente il lavoro del programmatore, che deve cercare di allineare correttamente i dati nello stack.
Con le **CPU** a **16** bit, **PUSH** e **POP** lavorano esclusivamente con operandi di tipo **WORD**; e' proibito l'uso di operandi di tipo **BYTE**.
Con le **CPU** a **32** bit, **PUSH** e **POP** lavorano esclusivamente con operandi di tipo **WORD** e **DWORD**; anche in questo caso, e' proibito l'uso di operandi di tipo **BYTE**.

16.3.1 L'istruzione PUSH

Con il mnemonico **PUSH**, si indica l'istruzione **PUSH operand onto the stack** (inserimento di un operando sulla cima dello stack). Questa istruzione richiede, esplicitamente, il solo operando **SRC**; infatti, l'operando **DEST** e' rappresentato, implicitamente, dalla locazione di memoria puntata da **SS:SP**.
Quando la **CPU** incontra una istruzione **PUSH** con operando di tipo **WORD**, esegue le seguenti operazioni:

1) sottrae **2** byte al registro **SP** per fare posto ad una nuova **WORD** nello stack;
2) copia la **WORD** contenuta in **SRC**, nella locazione di memoria all'indirizzo **SS:SP**.

Se l'operando e' di tipo **DWORD**, il registro **SP** viene decrementato di **4** byte.

Come si puo' notare, l'istruzione **PUSH** esegue un vero e proprio trasferimento dati; nel caso, ad esempio, dell'operando sorgente **AX**, l'istruzione **PUSH** (dopo il decremento di **SP**) equivale a:

```
mov ss:[sp], ax
```

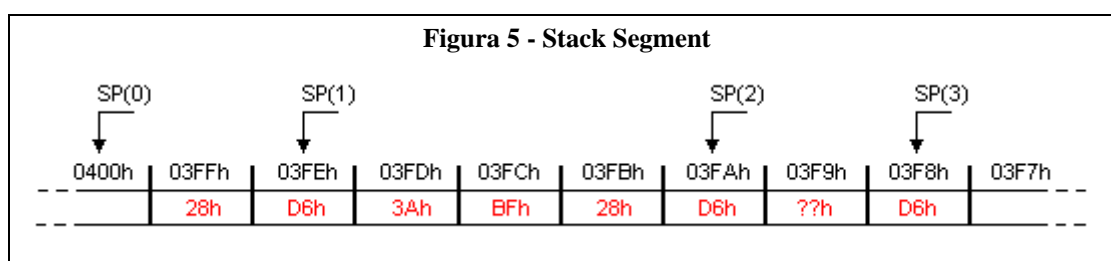
Naturalmente, questa istruzione ha un significato puramente simbolico; infatti, sappiamo che in modalita' reale, e' proibito dereferenziare il registro **SP**.

Per l'istruzione **PUSH**, sono permesse solo le seguenti forme:

```
PUSH     Reg16
PUSH     Reg32
PUSH     SegReg
PUSH     Mem16
PUSH     Mem32
PUSH     Imm
```

L'uso di un operando di tipo **Imm** e' permesso solo sulle **CPU 80186** e superiori; questo aspetto verra' trattato piu' avanti.

Vediamo una serie di esempi, nei quali supponiamo di partire con **SP=0400h**; questa situazione e' illustrata in Figura 5, dove il valore iniziale di **SP** e' rappresentato dal simbolo **SP(0)**.

Figura 5 - Stack Segment

In riferimento all'esempio di Figura 2, con **SP=0400h**, **Variabile16=28D6h** e con una direttiva **ASSUME** che associa **CODESEGM** a **CS**, possiamo scrivere l'istruzione:
```
push Variabile16
```
Quando la **CPU** incontra questa istruzione, esegue le seguenti operazioni:

1) sottrae **2** byte a **SP** e ottiene **SP=03FEh**;
2) legge il valore **28D6h** dall'indirizzo **CS:Variabile16**;
3) trasferisce **28D6h** nella locazione puntata da **SS:SP** (cioe', **SS:03FEh**).

In Figura 5, la nuova posizione di **SP** e' rappresentata dal simbolo **SP(1)**.

In riferimento all'esempio di Figura 3, con **SP=03FEh** e con **SS:BX** che punta a **Variabile32=3ABF28D6h**, possiamo scrivere l'istruzione:
```
push dword ptr ss:[bx]
```
Il segment override e' necessario, in modo che **BX** venga associato a **SS**; l'operatore **DWORD PTR** e' necessario, per specificare che l'operando **SRC** e' a **32** bit. Quando la **CPU** incontra questa istruzione, esegue le seguenti operazioni:

1) sottrae **4** byte a **SP** e ottiene **SP=03FAh**;
2) legge il valore **3ABF28D6h** dall'indirizzo **SS:BX**;
3) trasferisce **3ABF28D6h** nella locazione puntata da **SS:SP** (cioe', **SS:03FAh**).

In Figura 5, la nuova posizione di **SP** e' rappresentata dal simbolo **SP(2)**.

In riferimento all'esempio di Figura 1, con **SP=03FAh**, **Variabile8=D6h** e con una direttiva **ASSUME** che associa **DATASEGM** a **DS**, possiamo scrivere l'istruzione:
```
push Variabile8
```

In questo caso, l'assembler genera un messaggio di errore, per indicare che e' proibito l'uso di un operando a **8** bit; come dobbiamo comportarci in un caso del genere?
La soluzione consiste nel salvare **Variabile8** negli **8** bit meno significativi di un registro generale a **16** bit (preferibilmente **AX**); possiamo scrivere allora:

```
mov      al, Variabile8    ; al = Variabile8
push     ax                ; salva ax nello stack
```

In questo modo, ci riconduciamo al caso di un operando **SRC** a **16** bit; piu' avanti vedremo come si estrae dallo stack un operando a **8** bit. In presenza di una istruzione **PUSH** con operando **AX**, la **CPU** compie le seguenti operazioni:

1) sottrae **2** byte a **SP** e ottiene **SP=03F8h**;
2) legge il valore **??D6h** dal registro **AX**;
3) trasferisce **??D6h** nella locazione puntata da **SS:SP** (cioe', **SS:03F8h**).

In Figura 5, la nuova posizione di **SP** e' rappresentata dal simbolo **SP(3)**; osserviamo, inoltre, che all'indirizzo **SS:03F9h** sono stati salvati gli **8** bit piu' significativi di **AX** (in questo caso, il valore assunto da questi **8** bit non ha nessuna importanza).

16.3.2 Istruzione PUSH con operando di tipo Imm

Avendo a disposizione una **CPU 80186** o superiore, possiamo utilizzare **PUSH** anche con operandi di tipo **Imm**; consultando i manuali della **CPU**, ci accorgiamo che in questo caso, il codice macchina e' formato dall'**Opcode 011010s0**, seguito da un **Imm**. Il significato del bit indicato con **s** (**sign bit**) e' molto importante; infatti, questo bit indica alla **CPU** come deve essere trattato il valore **Imm** da inserire nello stack. Se **s=0**, la **CPU** non apporta nessuna modifica all'**Imm**; se, invece, **s=1**, la **CPU** effettua l'estensione del bit di segno dell'**Imm**.

Consideriamo la seguente dichiarazione:
`IMMEDIATO = +25 ; = 19h = 00011001b`
A questo punto, possiamo scrivere:
`push IMMEDIATO`
Quando l'assembler incontra questa istruzione, genera il codice macchina:
`01101010b 00011001b = 6Ah 19h`
Quando la **CPU** incontra questo codice macchina, vede che **s=1**, per cui deve estendere a **16** bit il valore **19h**, attraverso il bit di segno; il valore che viene inserito nello stack e' quindi:
`0000000000011001b = 0019h`

Consideriamo la seguente dichiarazione:
`IMMEDIATO = -25 ; = E7h = 11100111b`
A questo punto, possiamo scrivere:
`push IMMEDIATO`
Quando l'assembler incontra questa istruzione, genera il codice macchina:
`01101010b 11100111b = 6Ah E7h`
Quando la **CPU** incontra questo codice macchina, vede che **s=1**, per cui deve estendere a **16** bit il valore **E7h**, attraverso il bit di segno; il valore che viene inserito nello stack e' quindi:
`1111111111100111b = FFE7h`

Consideriamo la seguente dichiarazione:
`IMMEDIATO = +40950 ; = 9FF6h = 1001111111110110b`
A questo punto, possiamo scrivere:
`push IMMEDIATO`

Quando l'assembler incontra questa istruzione, genera il codice macchina:
`01101000b 1001111111110110b = 68h 9FF6h`
Quando la **CPU** incontra questo codice macchina, vede che **s=0**, per cui non deve modificare il valore **9FF6h**; il valore che viene inserito nello stack e' quindi:
`1001111111110110b = 9FF6h`

Consideriamo la seguente dichiarazione:
`IMMEDIATO = -12850 ; = CDCEh = 1100110111001110b`
A questo punto, possiamo scrivere:
`push IMMEDIATO`
Quando l'assembler incontra questa istruzione, genera il codice macchina:
`01101000b 1100110111001110b = 68h CDCEh`
Quando la **CPU** incontra questo codice macchina, vede che **s=0**, per cui non deve modificare il valore **CDCEh**; il valore che viene inserito nello stack e' quindi:
`1100110111001110b = CDCEh`

Consideriamo la seguente dichiarazione:
`IMMEDIATO = -25 ; = E7h = 11100111b`
A questo punto, possiamo scrivere:
`push dword ptr IMMEDIATO`
Quando l'assembler incontra questa istruzione, genera il codice macchina:
`01100110b 01101010b 11100111b = 66h 6Ah E7h`
Quando la **CPU** incontra questo codice macchina, vede che **s=1**, per cui deve estendere a **32** bit (prefisso **66h**) il valore **E7h**, attraverso il bit di segno; il valore che viene inserito nello stack e' quindi:
`11111111111111111111111111100111b = FFFFFFE7h`

Consideriamo, infine, la seguente dichiarazione:
`IMMEDIATO = -300000 ; = FFFB6C20h = 11111111111110110110110000100000b`
A questo punto, possiamo scrivere:
`push IMMEDIATO`
Quando l'assembler incontra questa istruzione, genera il codice macchina:
`01100110b 01101000b 11111111111110110110110000100000b = 66h 68h FFFB6C20h`
Quando la **CPU** incontra questo codice macchina, vede che **s=0**, per cui non deve modificare il valore **FFFB6C20h**; il valore che viene inserito nello stack e' quindi:
`11111111111110110110110000100000b = FFFB6C20h`

16.3.3 L'istruzione POP

Con il mnemonico **POP**, si indica l'istruzione **POP a value from the stack** (estrazione di un valore dalla cima dello stack). Questa istruzione richiede, esplicitamente, il solo operando **DEST**; infatti, l'operando **SRC** e' rappresentato, implicitamente, dalla locazione di memoria puntata da **SS:SP**.
Quando la **CPU** incontra una istruzione **POP** con operando di tipo **WORD**, esegue le seguenti operazioni:

1) copia in **DEST** la **WORD** che si trova nella locazione di memoria all'indirizzo **SS:SP**;
2) somma **2** byte al registro **SP** per recuperare lo spazio che si e' liberato nello stack.

Se l'operando e' di tipo **DWORD**, il registro **SP** viene incrementato di **4** byte.

Come si puo' notare, anche l'istruzione **POP** esegue un vero e proprio trasferimento dati; nel caso, ad esempio, dell'operando destinazione **AX**, l'istruzione **POP** (prima dell'incremento di **SP**) equivale a:
```
mov ax, ss:[sp]
```

Come al solito, questa istruzione ha un significato puramente simbolico; infatti, sappiamo che in modalita' reale, e' proibito dereferenziare il registro **SP**.

Per l'istruzione **POP**, sono permesse solo le seguenti forme:

```
POP      Reg16
POP      Reg32
POP      SegReg
POP      Mem16
POP      Mem32
```

L'istruzione **POP** con operando di tipo **Imm** non ha nessun senso; infatti, non e' possibile estrarre un valore dallo stack e metterlo in un **Imm**.
Nel caso dell'istruzione **POP** con operando di tipo **SegReg**, e' proibito scrivere:
```
POP CS
```
Abbiamo gia' visto, infatti, che solo la **CPU** puo' modificare il contenuto di **CS**; in un prossimo capitolo, vedremo che il caricamento in **CS** di una **WORD** estratta dallo stack, viene svolto dalla istruzione **RET** (ritorno da una procedura).
Prima di eseguire l'istruzione:
```
POP SS
```
la **CPU** disabilita automaticamente, sia le interruzioni mascherabili, sia le **NMI**; tutte le interruzioni vengono automaticamente ripristinate subito dopo l'esecuzione dell'istruzione successiva (che spesso e' un'altra **POP** con operando **SP** o **ESP**).

Ripetiamo ora in senso inverso, i primi tre esempi presentati per l'istruzione **PUSH**; partiamo quindi con il registro **SP** che contiene il valore **03F8h**. Questa situazione e' illustrata in Figura 5, dove il valore iniziale di **SP** e' rappresentato dal simbolo **SP(3)**; possiamo dire quindi che in questo momento, la cima dello stack si trova all'indirizzo **SS:03F8h**.

E' fondamentale ricordare che tutte le estrazioni dallo stack, devono essere effettuate in senso inverso rispetto agli inserimenti; alla fine, lo stack deve risultare perfettamente bilanciato (**SP=0400h**).
Prima di tutto, procediamo con l'estrazione del valore **D6h** che si trova all'indirizzo **SS:03F8h**; in riferimento alla Figura 5, con **SP=03F8h**, possiamo scrivere l'istruzione:
```
pop ax
```
Quando la **CPU** incontra questa istruzione, esegue le seguenti operazioni:

1) legge il valore **??D6h** dall'indirizzo **SS:SP** (cioe', **SS:03F8h**);
2) trasferisce **??D6h** nel registro **AX**;
3) somma **2** byte a **SP** e ottiene **SP=03FAh**.

In Figura 5, la nuova posizione di **SP** e' rappresentata dal simbolo **SP(2)**; a questo punto, se vogliamo rimettere **D6h** in **Variabile8**, non dobbiamo fare altro che scrivere (con una direttiva **ASSUME** che associa **DATASEGM** a **DS**):
```
mov Variabile8, al
```
Solamente gli **8** bit meno significativi di **AX**, contengono un valore valido (in questo caso, **D6h**); gli **8** bit piu' significativi di **AX**, contengono un valore casuale.

In riferimento alla Figura 5, con **SP=03FAh** e con una direttiva **ASSUME** che associa **STACKSEGM** a **SS**, possiamo scrivere l'istruzione:
```
pop Variabile32
```
Quando la **CPU** incontra questa istruzione, esegue le seguenti operazioni:

1) legge il valore **3ABF28D6h** dall'indirizzo **SS:SP** (cioe', **SS:03FAh**);
2) trasferisce **3ABF28D6h** nella locazione di memoria all'indirizzo **SS:Variabile32**;
3) somma **4** byte a **SP** e ottiene **SP=03FEh**.

In Figura 5, la nuova posizione di **SP** e' rappresentata dal simbolo **SP(1)**.

In riferimento alla Figura 5, con **SP=03FEh** e con **CS:(BX+DI+0002h)** che punta a **Variabile16**, possiamo scrivere l'istruzione:

```
pop word ptr cs:[bx+di+0002h]
```

Il segment override e' necessario, in modo che la **base BX** venga associata a **CS**; l'operatore **WORD PTR** e' necessario, per specificare che l'operando **DEST** e' a **16** bit. Quando la **CPU** incontra questa istruzione, esegue le seguenti operazioni:

1) legge il valore **28D6h** dall'indirizzo **SS:SP** (cioe', **SS:03FEh**);
2) trasferisce **28D6h** nella locazione di memoria all'indirizzo **CS:(BX+DI+0002h)**;
3) somma **2** byte a **SP** e ottiene **SP=0400h**.

In Figura 5, la nuova posizione di **SP** e' rappresentata dal simbolo **SP(0)**.

E' necessario ribadire che l'aspetto piu' importante nella gestione dello stack, riguarda il perfetto bilanciamento tra **PUSH** e **POP**; al termine di un programma, il numero di byte estratti con le istruzioni **POP**, deve essere pari al numero di byte inseriti con le istruzioni **PUSH** (questo discorso vale anche per una qualsiasi procedura, che puo' essere vista come un mini-programma).
Ricordiamo, inoltre, che se abbiamo scritto, ad esempio:

```
push Variabile32
```

non siamo obbligati poi a scrivere:

```
pop Variabile32
```

Possiamo anche scrivere:

```
pop ebx
```

L'importante e' che l'inserimento dei **32** bit di **Variabile32**, venga poi bilanciato dalla estrazione degli stessi **32** bit (che possiamo mettere dove ci pare).
Tutto cio' ci permette di utilizzare le istruzioni **PUSH** e **POP**, per eseguire rapidamente diverse operazioni; se, ad esempio, vogliamo copiare **DS** in **ES** senza utilizzare un registro generale, possiamo scrivere:

```
push     ds    ; salva il contenuto di DS nello stack
pop      es    ; poi lo estrae e lo mette in ES
```

Questo tipo di istruzioni compaiono molto frequentemente nei programmi **Assembly**; evidentemente, chi programma in questo modo, non sa' che (soprattutto con le vecchie **CPU**) si ottiene un notevole aumento della velocita' di esecuzione, con le istruzioni:

```
mov      ax, ds   ; AX = DS
mov      es, ax   ; ES = AX
```

Un'altra operazione che possiamo eseguire velocemente con **PUSH** e **POP**, consiste nel copiare due **WORD** in una **DWORD**; se, ad esempio, vogliamo copiare il contenuto di **BX** nei **16** bit piu' significativi di **EAX**, e il contenuto di **CX** nei **16** bit meno significativi di **EAX**, possiamo scrivere:

```
mov      bx, 3CBFh    ; bx = 3CBFh
mov      cx, 28F6h    ; cx = 28F6h

push     bx           ; salva il contenuto di BX nello stack
push     cx           ; salva il contenuto di CX nello stack
pop      eax          ; estrae 32 bit e li mette in EAX

mov      dx, 0400h    ; riga 4, colonna 0
call     writeHex32   ; visualizza eax = 3CBF28F6h
```

E' importante capire bene come vanno a disporsi in **EAX** i contenuti di **BX** e **CX**; a tale proposito, conviene sempre disegnare uno schema dello stack, come quello mostrato in Figura 5.

16.3.4 Effetti provocati da PUSH e POP sugli operandi e sui flags

L'esecuzione delle istruzioni **PUSH** e **POP**, modifica il contenuto del solo operando **DEST**; infatti, il vecchio contenuto di **DEST** viene sovrascritto dal contenuto di **SRC**. Il contenuto dell'operando **SRC** rimane inalterato; in relazione, pero', alla istruzione **PUSH**, si presenta un caso molto delicato, rappresentato da:
`push sp` o `push esp`
Il contenuto di **SP/ESP** che viene salvato nello stack, e' quello precedente o quello successivo al decremento dello stesso **SP/ESP**?
La risposta a questa domanda, varia da **CPU** a **CPU**; per analizzare i diversi casi, supponiamo di avere **SP=0400h**.
La **CPU 8086**, in presenza della precedente istruzione, esegue le seguenti operazioni:

1) sottrae **2** byte a **SP** e ottiene **SP=03FEh**;
2) legge il valore **03FEh** presente nel registro **SP**;
3) trasferisce **03FEh** nella locazione puntata da **SS:SP** (cioe', **SS:03FEh**).

I progettisti delle **CPU**, si sono resi conto che questa situazione non e' del tutto corretta; infatti, appare piu' logico il salvataggio nello stack del valore di **SP** che precede il decremento. A partire quindi dalla **80286**, in presenza della precedente istruzione la **CPU** esegue le seguenti operazioni:

1) memorizza il valore **0400h** contenuto in **SP**;
2) sottrae **2** byte a **SP** e ottiene **SP=03FEh**;
3) trasferisce **0400h** nella locazione puntata da **SS:SP** (cioe', **SS:03FEh**).

In sostanza, indicando con **Temp16** un registro interno a **16** bit della **CPU**, possiamo simulare il procedimento appena descritto, con le seguenti pseudo-istruzioni:

```
mov      Temp16, sp
push     Temp16
```

Il problema appena descritto in relazione alla istruzione **PUSH**, si presenta anche per l'istruzione:
`pop sp` o `pop esp`
Il valore caricato in **SP/ESP**, e' quello precedente o quello successivo all'incremento dello stesso **SP/ESP**?

Fortunatamente, in questo caso il problema e' stato subito risolto per tutti i modelli di **CPU** della famiglia **80x86**; per analizzare questo caso, supponiamo di avere **SP=03FEh**, e **SS:[SP]=13B8h**. In presenza della precedente istruzione, la **CPU** esegue le seguenti operazioni:

1) memorizza il valore **13B8h** presente all'indirizzo **SS:SP** (cioe', **SS:03FEh**);
2) somma **2** byte a **SP** e ottiene **SP=0400h**;
3) trasferisce **13B8h** nel registro **SP**.

In sostanza, indicando con **Temp16** un registro interno a **16** bit della **CPU**, possiamo simulare il procedimento appena descritto, con le seguenti pseudo-istruzioni:

```
mov      Temp16, ss:[sp]
add      sp, 2
mov      sp, Temp16
```

Come si puo' notare, quando l'operando **DEST** di **POP** e' **SP** (o **ESP**), le fasi **2** (trasferimento dati) e **3** (incremento di **SP/ESP**) vengono scambiate tra loro; naturalmente, si da' per scontato che nell'usare l'istruzione **POP** con operandi del tipo **SP/ESP**, **DS**, **ES**, **SS**, etc, il programmatore sappia esattamente cio' che sta facendo.

L'esecuzione delle istruzioni **PUSH** e **POP**, non modifica nessuno dei campi presenti nel **Flags Register**.

16.4 Le istruzioni PUSHA, PUSHAD, POPA, POPAD

Una situazione che si presenta molto spesso in **Assembly**, consiste nella chiamata di una procedura, la quale modifica uno o piu' registri, senza preservarne il contenuto originale; quando la procedura termina, il "chiamante" riottiene il controllo, e si ritrova con diversi registri, il cui contenuto non e' piu' quello precedente alla chiamata della procedura stessa.
Supponiamo, ad esempio, che il programma principale stia utilizzando i registri **AX** e **BX**, per contenere informazioni importanti; ad un certo punto, lo stesso programma principale deve chiamare una procedura di nome **Proc1**, la quale modifica **AX** e **BX** senza preservarne il contenuto originale. Se non vogliamo perdere il contenuto originale di **AX** e **BX**, possiamo scrivere le seguenti istruzioni:

```
push     ax       ; salva il contenuto di ax
push     bx       ; salva il contenuto di bx
call     Proc1    ; chiama la procedura
pop      bx       ; ripristina bx
pop      ax       ; ripristina ax
```

Naturalmente, questo procedimento funziona solo se, all'interno di **Proc1**, non vengono commessi errori nella gestione dello stack.
Questo modo di programmare, appare piuttosto discutibile, in quanto, per ogni chiamata di **Proc1**, dobbiamo inserire due istruzioni **PUSH** e due istruzioni **POP**; come e' facile intuire, si tratta di una situazione che, oltre ad accrescere le dimensioni del programma, ci espone anche ad una elevata probabilita' di commettere qualche errore.
La strada piu' logica e piu' sicura da seguire, consiste nel fare in modo che sia la procedura stessa a preservare il contenuto di tutti i registri che utilizza; in sostanza, tutte le necessarie istruzioni **PUSH** e **POP**, devono essere inserite all'interno della procedura.

In certi casi, e' fondamentale che una procedura preservi il contenuto di tutti i registri che utilizza; questa situazione si presenta, ad esempio, per le **ISR**.
Come sappiamo, quando arriva una richiesta di interruzione hardware, la **CPU** sospende temporaneamente il programma in esecuzione, e chiama l'opportuna **ISR**; le richieste di interruzione hardware, arrivano quindi in modo asincrono (cioe' non sincronizzato) rispetto al programma in esecuzione. Al termine della **ISR**, la **CPU** riavvia il programma precedentemente interrotto, il quale si aspetta di trovare intatti, tutti i registri che stava utilizzando; se la **ISR** non ha preservato il contenuto dei registri che ha utilizzato, si verifica quindi un sicuro crash.

L'utilizzo di un numero eccessivo (e ingiustificato) di istruzioni **PUSH** e **POP**, influisce negativamente sulle prestazioni di un programma; proprio per questo motivo, e' necessario limitare al massimo il loro impiego.
Nel caso in cui sia assolutamente necessario preservare il contenuto di numerosi registri generali, puo' rivelarsi vantaggioso (in termini di cicli macchina) l'impiego delle istruzioni **PUSHA**, **PUSHAD**, **POPA** e **POPAD**, disponibili a partire dalle **CPU 80186**; ciascuna di queste istruzioni, e' in grado di gestire ben **8** registri generali, sia a **16**, sia a **32** bit.
Le uniche forme lecite per queste istruzioni, sono le seguenti:

```
PUSHA
PUSHAD
POPA
POPAD
```

Come si puo' notare, tutte queste istruzioni devono essere utilizzate senza nessun operando esplicito; infatti, sia l'operando **SRC**, sia l'operando **DEST**, vengono stabiliti, implicitamente, dalla **CPU**.
Proprio per questo motivo, diventa determinante l'attributo **Dimensione** assegnato al segmento di programma nel quale sono presenti queste istruzioni; e' necessario quindi distinguere tra modalita' reale (attributo **USE16**), e modalita' protetta (attributo **USE32**).
In presenza dell'attributo **USE16**, le istruzioni **PUSHA** e **POPA** operano sui registri generali a **16** bit, mentre le istruzioni **PUSHAD** e **POPAD**, operano sui registri generali a **32** bit; in presenza, invece, dell'attributo **USE32**, tutte queste istruzioni operano, in ogni caso, sui registri generali a **32** bit.
Nel seguito, facciamo riferimento ad un programma **Assembly** destinato alla modalita' reale; tutti i segmenti di questo programma, sono quindi dotati di attributo **USE16**.

16.4.1 Le istruzioni PUSHA e PUSHAD

Con il mnemonico **PUSHA**, si indica l'istruzione **PUSH All general registers onto the stack** (inserimento sulla cima dello stack, del contenuto di tutti i registri generali); in presenza dell'istruzione:
`pusha`
la **CPU** compie le seguenti operazioni:

1) sottrae **8*2=16** byte al registro **SP** per fare posto a **8** nuove **WORD** nello stack;
2) salva nello stack il contenuto della seguente sequenza ordinata di registri generali:
`AX, CX, DX, BX, SP originale, BP, SI, DI`

In sostanza, l'istruzione **PUSHA** equivale a:

```
mov      Temp16, sp
push     ax
push     cx
push     dx
push     bx
push     Temp16
push     bp
push     si
push     di
```

Per rendere esplicito il fatto che vogliamo operare sui registri generali a **16** bit, il **TASM** fornisce l'istruzione **PUSHAW** (dove la **W** sta per **WORD size**), che e' un alias per **PUSHA**; l'istruzione **PUSHAW**, non e' quindi disponibile con il **MASM**.

Con il mnemonico **PUSHAD**, si indica l'istruzione **PUSH All general registers (Dword size) onto the stack** (inserimento sulla cima dello stack, del contenuto di tutti i registri generali a **32** bit); in presenza dell'istruzione:
`pushad`
la **CPU** compie le seguenti operazioni:

1) sottrae **8*4=32** byte al registro **SP** per fare posto a **8** nuove **DWORD** nello stack;
2) salva nello stack il contenuto della seguente sequenza ordinata di registri generali:
`EAX, ECX, EDX, EBX, ESP originale, EBP, ESI, EDI`

In sostanza, l'istruzione **PUSHAD** equivale a:

```
mov      Temp32, esp
push     eax
push     ecx
push     edx
push     ebx
push     Temp32
push     ebp
push     esi
push     edi
```

Le istruzioni **PUSHA** e **PUSHAD** hanno l'identico codice macchina **01100000b** (**60h**); in modalita' reale, per permettere alla **CPU** di distinguere **PUSHAD** da **PUSHA**, l'assembler inserisce il prefisso **66h**.

16.4.2 Le istruzioni POPA e POPAD

Con il mnemonico **POPA**, si indica l'istruzione **POP All general registers from the stack** (estrazione dalla cima dello stack, di valori da trasferire in tutti i registri generali); in presenza dell'istruzione:
`popa`
la **CPU** compie le seguenti operazioni:

1) estrae **8 WORD** dalla cima dello stack, e le trasferisce nella seguente sequenza ordinata di registri generali:
`DI, SI, BP, --, BX, DX, CX, AX`
2) somma **8*2=16** byte al registro **SP**, per recuperare lo spazio che si e' liberato nello stack.

In sostanza, l'istruzione **PUSHA** equivale a:

```
pop      di
pop      si
pop      bp
add      sp, 2
pop      bx
pop      dx
pop      cx
pop      ax
```

Come si puo' notare, la sequenza di estrazione con **POPA**, e' ovviamente invertita rispetto alla sequenza di inserimento con **PUSHA**; la quarta **WORD**, destinata a **SP**, viene ignorata (cioe', **SP** viene direttamente incrementato di **2** byte senza che venga effettuata nessuna estrazione).

Per rendere esplicito il fatto che vogliamo operare sui registri generali a **16** bit, il **TASM** fornisce l'istruzione **POPAW** (dove la **W** sta per **WORD size**), che e' un alias per **POPA**; l'istruzione **POPAW**, non e' quindi disponibile con il **MASM**.

Con il mnemonico **POPAD**, si indica l'istruzione **POP All general registers (Dword size) from the stack** (estrazione dalla cima dello stack, di valori da trasferire in tutti i registri generali a **32** bit); in presenza dell'istruzione:
`popad`
la **CPU** compie le seguenti operazioni:

1) estrae **8 DWORD** dalla cima dello stack, e le trasferisce nella seguente sequenza ordinata di registri generali:
`EDI, ESI, EBP, ---, EBX, EDX, ECX, EAX`
2) somma **8*4=32** byte al registro **SP**, per recuperare lo spazio che si e' liberato nello stack.

In sostanza, l'istruzione **PUSHAD** equivale a:

```
pop      edi
pop      esi
pop      ebp
add      esp, 4
pop      ebx
pop      edx
pop      ecx
pop      eax
```

Anche in questo caso, si nota che la sequenza di estrazione con **POPAD**, e' ovviamente invertita rispetto alla sequenza di inserimento con **PUSHAD**; la quarta **DWORD**, destinata a **ESP**, viene ignorata (cioe', **ESP** viene direttamente incrementato di **4** byte senza che venga effettuata nessuna estrazione).

Le istruzioni **POPA** e **POPAD** hanno l'identico codice macchina **01100001b** (**61h**); in modalita' reale, per permettere alla **CPU** di distinguere **PUSHAD** da **PUSHA**, l'assembler inserisce il prefisso **66h**.

16.4.3 Effetti provocati da PUSHA, PUSHAD, POPA e POPAD, sugli operandi e sui flags

Le istruzioni **PUSHA**, **PUSHAD**, **POPA** e **POAD**, modificano il solo operando **DEST**, che viene sovrascritto dal contenuto dell'operando **SRC**; il contenuto dell'operando **SRC** rimane inalterato.
Come accade per **PUSH**, anche per **PUSHA** e **PUSHAD** si pone il problema legato alla presenza del registro **SP/ESP**; questo problema e' stato risolto, salvando nello stack il valore originale di **SP/ESP**, cioe' il valore che precede il decremento di **SP/ESP**.
In relazione, invece, alle istruzioni **POPA** e **POPAD**, il problema non si pone; infatti, abbiamo appena visto che il valore estratto dallo stack e destinato a **SP/ESP**, viene scartato.

L'esecuzione delle istruzioni **PUSHA**, **PUSHAD**, **POPA** e **POPAD**, non modifica nessuno dei campi presenti nel **Flags Register**.

16.5 Le istruzioni PUSHF, PUSHFD, POPF, POPFD

Se vogliamo gestire velocemente l'intero contenuto del registro dei flags, possiamo servirci delle istruzioni **PUSHF**, **PUSHFD**, **POPF** e **POPFD**; le uniche forme lecite per queste istruzioni, sono le seguenti:

```
PUSHF
PUSHFD
POPF
POPFD
```

Come si puo' notare, tutte queste istruzioni devono essere utilizzate senza nessun operando esplicito; infatti, sia l'operando **SRC**, sia l'operando **DEST**, vengono stabiliti, implicitamente, dalla **CPU**.
Anche in questo caso quindi, diventa determinante l'attributo **Dimensione** assegnato al segmento di programma nel quale sono presenti queste istruzioni; e' necessario cioe', distinguere tra modalita' reale (attributo **USE16**), e modalita' protetta (attributo **USE32**).
In presenza dell'attributo **USE16**, le istruzioni **PUSHF** e **POPF** operano sul registro **FLAGS**, mentre le istruzioni **PUSHFD** e **POPFD**, operano sul registro **EFLAGS**; in presenza, invece, dell'attributo **USE32**, tutte queste istruzioni operano, in ogni caso, sul registro **EFLAGS**.
Nel seguito, facciamo riferimento ad un programma **Assembly** destinato alla modalita' reale.

Ricordiamo che il registro **FLAGS** occupa **16** bit, e assume la struttura mostrata in Figura 6; il registro **EFLAGS** occupa **32** bit, e rappresenta l'estensione del registro **FLAGS**.

Figura 6 - Registro FLAGS

Flag	-	-	-	-	OF	DF	IF	TF	SF	ZF	-	AF	-	PF	-	CF
Posizione	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

16.5.1 Le istruzioni PUSHF e PUSHFD

Con il mnemonico **PUSHF**, si indica l'istruzione **PUSH Flags register onto the stack** (inserimento sulla cima dello stack, del contenuto a **16** bit del registro **FLAGS**); in presenza dell'istruzione:

```
pushf
```

la **CPU** compie le seguenti operazioni:

1) sottrae **2** byte al registro **SP** per fare posto a una nuova **WORD** nello stack;
2) salva il contenuto del registro **FLAGS**, nella locazione di memoria all'indirizzo **SS:SP**.

Per rendere esplicito il fatto che vogliamo operare sul registro **FLAGS** a **16** bit, il **TASM** fornisce l'istruzione **PUSHFW** (dove la **W** sta per **WORD size**), che e' un alias per **PUSHF**; l'istruzione **PUSHFW**, non e' quindi disponibile con il **MASM**.

Con il mnemonico **PUSHFD**, si indica l'istruzione **PUSH eFlags register onto the stack (Dword size)** (inserimento sulla cima dello stack, del contenuto a **32** bit del registro **EFLAGS**); in presenza dell'istruzione:

```
pushfd
```

la **CPU** compie le seguenti operazioni:

1) sottrae **4** byte al registro **SP** per fare posto a una nuova **DWORD** nello stack;
2) salva il contenuto del registro **EFLAGS**, nella locazione di memoria all'indirizzo **SS:SP**.

Le istruzioni **PUSHF** e **PUSHFD** hanno l'identico codice macchina **10011100b** (**9Ch**); in modalita' reale, per permettere alla **CPU** di distinguere **PUSHFD** da **PUSHF**, l'assembler inserisce il prefisso **66h**.

Supponiamo, ad esempio, di voler effettuare una addizione, per verificarne gli effetti prodotti sul registro **FLAGS**; possiamo scrivere allora le seguenti istruzioni:

```
; verifica dello stato attuale dei flags

pushf                ; salva il contenuto di FLAGS nello stack
pop      ax          ; e lo copia in AX
mov      dx, 0400h   ; riga 4, colonna 0
call     writeBin16  ; visualizza FLAGS in binario

; (-32768) + (-1) = 8000h + FFFFh = 7FFFh = +32767 (overflow)

mov      ax, -32768  ; ax = -32768
add      ax, -1      ; ax = (-32768) + (-1) = +32767
mov      dx, 0500h   ; riga 5, colonna 0
call     writeSdec16 ; verifica il risultato

; verifica dello stato attuale dei flags

pushf                ; salva il contenuto di FLAGS nello stack
pop      ax          ; e lo copia in AX
mov      dx, 0600h   ; riga 6, colonna 0
call     writeBin16  ; visualizza FLAGS in binario
```

Per il corretto funzionamento di questo esempio, e' fondamentale il fatto che le istruzioni **MOV**, **POP** e **CALL**, non modificano nessun campo del registro **FLAGS**; inoltre, le procedure **writeBin16** e **writeSdec16**, preservano il contenuto del registro **FLAGS**.

16.5.2 Le istruzioni POPF e POPFD

Con il mnemonico **POPF**, si indica l'istruzione **POP from stack into Flags register** (estrazione dalla cima dello stack, di una **WORD** da trasferire nel registro **FLAGS**); in presenza dell'istruzione:

```
popf
```

la **CPU** compie le seguenti operazioni:

1) estrae **1 WORD** dalla cima dello stack, e la trasferisce nel registro **FLAGS**;
2) somma **2** byte al registro **SP**, per recuperare lo spazio che si e' liberato nello stack.

Per rendere esplicito il fatto che vogliamo operare sul registro **FLAGS** a **16** bit, il **TASM** fornisce l'istruzione **POPFW** (dove la **W** sta per **WORD size**), che e' un alias per **POPF**; l'istruzione **POPFW**, non e' quindi disponibile con il **MASM**.

Con il mnemonico **POPFD**, si indica l'istruzione **POP from stack into eFlags register (Dword size)** (estrazione dalla cima dello stack, di una **DWORD** da trasferire nel registro **EFLAGS**); in presenza dell'istruzione:

```
popfd
```

la **CPU** compie le seguenti operazioni:

1) estrae **1 DWORD** dalla cima dello stack, e la trasferisce nel registro **EFLAGS**;
2) somma **4** byte al registro **SP**, per recuperare lo spazio che si e' liberato nello stack.

Le istruzioni **POPF** e **POPFD** hanno l'identico codice macchina **10011101b** (**9Dh**); in modalita' reale, per permettere alla **CPU** di distinguere **POPFD** da **POPF**, l'assembler inserisce il prefisso **66h**.

16.5.3 Effetti provocati da PUSHF, PUSHFD, POPF e POPFD, sugli operandi e sui flags

Le istruzioni **PUSHF**, **PUSHFD**, **POPF** e **POPFD**, modificano il solo operando **DEST**, che viene sovrascritto dal contenuto dell'operando **SRC**; il contenuto dell'operando **SRC** rimane inalterato.

L'esecuzione delle istruzioni **PUSHF** e **PUSHFD**, non modifica nessuno dei campi presenti nel registro **FLAGS/EFLAGS**.
In modalita' reale, l'esecuzione delle istruzioni **POPF** e **POPFD**, modifica tutti i campi presenti nel solo registro **FLAGS** (cioe', i campi visibili in Figura 6); i campi presenti nei **16** bit piu' significativi di **EFLAGS**, rimangono inalterati.

16.6 Le istruzioni LAHF e SAHF

Sempre in relazione alla gestione del **Flags Register**, sono disponibili anche le due istruzioni **LAHF** e **SAHF**; le uniche forme permesse per queste istruzioni, sono:

```
LAHF
SAHF
```

Come si puo' notare, queste istruzioni devono essere utilizzate senza nessun operando esplicito; infatti, sia l'operando **SRC**, sia l'operando **DEST**, vengono stabiliti, implicitamente, dalla **CPU**.
Come al solito, diventa determinante l'attributo **Dimensione** assegnato al segmento di programma nel quale sono presenti queste istruzioni; se l'attributo e' **USE16**, viene coinvolto il registro **FLAGS**, mentre se l'attributo e' **USE32**, viene coinvolto il registro **EFLAGS**.

16.6.1 L'istruzione LAHF

Con il mnemonico **LAHF**, si indica l'istruzione **Load status Flags into AH register** (trasferimento nel registro **AH**, degli **8** bit meno significativi del registro **FLAGS**); in presenza dell'istruzione:

```
lahf
```

la **CPU** legge gli **8** bit meno significativi del registro **FLAGS**, e li trasferisce nel registro **AH**; analizzando la Figura 6, possiamo dire che dopo l'esecuzione di questa istruzione, il registro **AH** assume la struttura mostrata in Figura 7.

Figura 7 - Registro AH

Flag	SF	ZF	-	AF	-	PF	-	CF
Posizione	7	6	5	4	3	2	1	0

16.6.2 L'istruzione SAHF

Con il mnemonico **SAHF**, si indica l'istruzione **Store AH into Flags** (trasferimento del contenuto del registro **AH**, negli **8** bit meno significativi del registro **FLAGS**); in presenza dell'istruzione:

```
sahf
```

la **CPU** legge il contenuto del registro **AH**, e lo trasferisce negli **8** bit meno significativi del registro **FLAGS**; si da' per scontato il fatto che il contenuto di **AH**, abbia un senso logico.

In genere, quando si ha la necessita' di accedere in lettura/scrittura al **Flags Register**, si utilizzano istruzioni come **PUSHF**, **POPF**, etc; proprio per questo motivo, le due istruzioni **LAHF** e **SAHF** vengono utilizzate con minore frequenza. Un caso particolare e' rappresentato dalla comparazione tra due numeri reali effettuata dal coprocessore matematico; come viene spiegato in un apposito capitolo della sezione **Assembly Avanzato**, in questo caso **LAHF** e **SAHF** ricoprono un ruolo importante nel determinare il risultato della comparazione stessa.

16.6.3 Effetti provocati da LAHF e SAHF sugli operandi e sui flags

Le istruzioni **LAHF** e **SAHF**, modificano il solo operando **DEST**, che viene sovrascritto dal contenuto dell'operando **SRC**; il contenuto dell'operando **SRC** rimane inalterato.

L'esecuzione dell'istruzione **LAHF**, non modifica nessuno dei campi presenti nel registro **FLAGS/EFLAGS**.
In modalita' reale, l'esecuzione dell'istruzione **SAHF** modifica tutti i campi presenti negli **8** bit meno significativi del registro **FLAGS**; come si nota dalla Figura 7, i flags interessati sono, **SF**, **ZF**, **AF**, **PF** e **CF**.

16.7 L'istruzione LEA

Consideriamo la seguente istruzione:
```
mov ax, bx + offset Variabile8
```
Incontrando questa istruzione, l'assembler genera un messaggio di errore, per indicare che non e' in grado di calcolare la somma tra il contenuto del registro **BX** e l'offset di **Variabile8**; infatti, solo nella fase di esecuzione di un programma, e' possibile conoscere il contenuto del registro **BX**, o di qualsiasi altro registro della **CPU**.

In sostanza, il calcolo della precedente somma, puo' essere svolto solo dalla **CPU**, durante la fase di esecuzione del programma; a tale proposito, la **CPU** stessa ci mette a disposizione una potente istruzione, chiamata **LEA**.
Con il mnemonico **LEA**, si indica l'istruzione **Load Effective Address** (trasferimento nel registro **DEST**, dell'**effective address** dell'operando **SRC**); le uniche forme lecite per questa istruzione, sono le seguenti (la sigla **EA** sta per **effective address**):

```
LEA   Reg16, EA16
LEA   Reg32, EA16
LEA   Reg16, EA32
LEA   Reg32, EA32
```

Come abbiamo visto nel Capitolo 11, tutte le forme di indirizzamento permesse dalle **CPU** della famiglia **80x86**, vengono definite **effective address**; l'operando **SRC** dell'istruzione **LEA**, deve essere, appunto, uno tra gli **effective address** mostrati nelle Figure 8, 15 e 18 del Capitolo 11.
Il compito di **LEA**, e' quello di calcolare l'**effective address** specificato dall'operando **SRC**; il risultato di questo calcolo, viene trasferito nell'operando **DEST**, che deve essere un registro a **16** o a **32** bit.
Come sappiamo, in modalita' reale l'**effective address** rappresenta la componente **Offset** di un indirizzo logico **Seg:Offset**; l'istruzione **LEA**, calcola il solo **effective address** di un indirizzo logico, e non tiene quindi conto della eventuale presenza della componente **Seg** dell'indirizzo stesso.
In sostanza, nel caso di una istruzione del tipo:
```
lea ax, ss:[bx+di+0004h]
```
la presenza del registro **SS** e' del tutto superflua; nel registro **AX**, viene trasferito il risultato della somma:
```
BX + DI + 0004h
```
Appare evidente il fatto che il risultato di questa somma, e' del tutto indipendente dalla presenza o meno di un registro di segmento.

Il codice macchina dell'istruzione **LEA** e' formato dal campo **Opcode 10001101b** (**8Dh**), dal campo **mod_reg_r/m** e da un eventuale **Disp**; analizziamo i **4** possibili casi che si possono presentare.

16.7.1 Effective address a 16 bit e registro DEST a 16 bit

Poniamo **BX=0006h**, **SI=0004h**, e scriviamo l'istruzione:
```
lea cx, [bx+si+0002h]
```
Il registro destinazione e' **CX**, per cui **reg=001b**; l'operando sorgente e' un **effective address** del tipo **[BX+SI+Disp8]**, per cui **mod=01b**, **r/m=000b**. Otteniamo quindi:
```
mod_reg_r/m = 01001000b = 48h
```
Il **Disp8** e':
```
Disp8 = 00000010b = 02h
```
L'assembler genera quindi il codice macchina:
```
10001101b 01001000b 00000010b = 8Dh 48h 02h
```
In presenza di questo codice macchina, la **CPU** calcola:

```
BX + SI + 02h = 0006h + 0004h + 0002h = 000Ch
```
Il risultato **000Ch** viene quindi trasferito nei **16** bit del registro **CX**.

Nel caso particolare di una istruzione del tipo:
```
lea bx, Variabile8
```
il codice macchina e' formato dal campo **Opcode**, dal campo **mod_reg_r/m** e da un **Disp16**; il **TASM** genera, invece, il codice macchina dell'istruzione:
```
mov bx, offset Variabile8
```
In effetti, le due istruzioni sono del tutto equivalenti, ma con l'istruzione **MOV**, il codice macchina richiede **1** byte in meno; infatti, il trasferimento dati da **Imm16** a **Reg16**, ha un codice macchina formato dal campo **Opcode** e da un **Imm16**.

16.7.2 Effective address a 16 bit e registro DEST a 32 bit

In riferimento all'esempio di Figura 1, osserviamo che **Variabile8** si trova all'offset **0008h** del segmento **DATASEGM**; in presenza di una direttiva **ASSUME** che associa **DATASEGM** a **DS**, e con **BX=0020h**, scriviamo l'istruzione:
```
lea edx, Variabile8[bx]
```
Il registro destinazione e' **EDX**, per cui **reg=010b**; l'operando sorgente, cioe' lo "strano" simbolo **Variabile8[BX]**, non e' altro che un banalissimo **effective address** del tipo **[BX+Disp16]** (che puo' essere scritto anche come **[BX+Variabile8]**), per cui **mod=10b**, **r/m=111b**. Otteniamo quindi:
```
mod_reg_r/m = 10010111b = 97h
```
Il **Disp16** e':
```
Disp16 = 0000000000001000b = 0008h
```
L'operando **DEST** e' a **32** bit, per cui si rende necessario il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
01100110b 10001101b 10010111b 0000000000001000b = 66h 8Dh 97h 0008h
```
In presenza di questo codice macchina, la **CPU** calcola:
```
BX + 0008h = 0020h + 0008h = 0028h
```
Il valore **0028h** viene esteso a **32** bit con degli zeri, e si ottiene **00000028h**; il risultato **00000028h** viene quindi trasferito nei **32** bit del registro **EDX**.

In assenza della direttiva **ASSUME** che associa **DATASEGM** a **DS**, l'assembler non genera nessun messaggio di errore; e' ovvio infatti, che il calcolo dell'offset di **Variabile8** non ha niente a che vedere con il **SegReg** a cui viene associato **DATASEGM**.

16.7.3 Effective address a 32 bit, e registro DEST a 16 bit

Poniamo **EBP=00008BFAh**, e scriviamo l'istruzione:
```
lea ax, [ebp+00001FB8h]
```
Il registro destinazione e' **AX**, per cui **reg=000b**; l'operando sorgente e' un **effective address** del tipo **[EBP+Disp32]**, per cui **mod=10b**, **r/m=101b**. Otteniamo quindi:
```
mod_reg_r/m = 10000101b = 85h
```
Il **Disp32** e':
```
Disp32 = 00000000000000000001111110111000b = 00001FB8h
```
L'indirizzo **SRC** e' a **32** bit, per cui si rende necessario il prefisso **67h**; l'assembler genera quindi il codice macchina:
```
01100111b 10001101b 10000101b 00000000000000000001111110111000b = 67h 8Dh 85h
00001FB8h
```
In presenza di questo codice macchina, la **CPU** calcola:
```
EBP + 00001FB8h = 00008BFAh + 00001FB8h = 0000ABB2h
```
I **16** bit meno significativi del risultato, e cioe' **ABB2h**, vengono quindi trasferiti nei **16** bit del registro **AX**.

16.7.4 Effective address a 32 bit, e registro DEST a 32 bit

Poniamo **ECX=0000A1B8h**, **EDI=00000020h**, e scriviamo l'istruzione:

```
lea ebx, [ecx+(edi*4)+00002C8Eh]
```

Il registro destinazione e' **EBX**, per cui **reg=011b**; l'operando sorgente e' un **effective address** del tipo **[ECX+(indice scalato)+Disp32]**, per cui **mod=10b**, **r/m=100b**. Otteniamo quindi:

```
mod_reg_r/m = 10011100b = 9Ch
```

Il **fattore di scala** e' **4**, il registro **base** e' **ECX** e il registro **indice** e' **EDI**, per cui **scale=10b**, **index=111b**, **base=001b**; otteniamo quindi:

```
S.I.B. = 10111001b = B9h
```

Il **Disp32** e':

```
Disp32 = 00000000000000000010110010001110b = 00002C8Eh
```

L'operando **DEST** e' a **32** bit, per cui si rende necessario il prefisso **66h**; l'indirizzo **SRC** e' a **32** bit, per cui si rende necessario il prefisso **67h**. L'assembler genera quindi il codice macchina:

```
01100110b 01100111b 10001101b 10011100b 10111001b
00000000000000000010110010001110b =
66h 67h 8Dh 9Ch B9h 00002C8Eh
```

In presenza di questo codice macchina, la **CPU** calcola:

```
ECX + (EDI * 4) + 00002C8Eh = 0000A1B8h + (00000020h * 4) + 00002C8Eh =
0000CEC6h
```

Il risultato **0000CEC6h** viene quindi trasferito nei **32** bit del registro **EBX**.

16.7.5 Effetti provocati da LEA sugli operandi e sui flags

L'esecuzione dell'istruzione **LEA**, modifica il contenuto del solo operando **DEST**, che viene sovrascritto dal contenuto di **SRC**; il contenuto dell'operando **SRC** rimane inalterato.
Anche per **LEA**, bisogna prestare attenzione ai casi del tipo:

```
lea bx, [bx+0008h]
```

Supponiamo di avere **BX=0020h**, e **DS:[BX+0008h]=DS:[0028h]=28FAh**; dopo l'esecuzione della precedente istruzione, il contenuto **28FAh** della locazione di memoria all'indirizzo **DS:0028h** rimane inalterato. In relazione, invece, al registro **BX**, accade che:

***** prima dell'esecuzione di **LEA**, **BX=0020h**;
***** dopo l'esecuzione di **LEA**, **BX=0020h+0008h=0028h**.

Di conseguenza, **DS:(BX+0008h)** non rappresenta piu' l'indirizzo **DS:0028h**, ma bensi' **DS:0030h**.

L'esecuzione dell'istruzione **LEA**, non modifica nessuno dei campi presenti nel **Flags Register**.

16.8 Le istruzioni LDS, LES, LFS, LGS, LSS

Nota importante.
Come sappiamo, un indirizzo **FAR** e' formato da una coppia di valori, indicati simbolicamente con **Seg:Offset**; la componente **Offset**, rappresenta uno spiazzamento, e occupa, **16** bit in modalita' reale, e **32** bit in modalita' protetta. La componente **Seg** occupa sempre **16** bit, e il suo significato varia in base alla modalita' operativa; in modalita' reale, **Seg** rappresenta un segmento di memoria, mentre in modalita' protetta, **Seg** rappresenta un cosiddetto **selettore**.
Tutte le **CPU** della famiglia **80x86**, seguono una importantissima convenzione per la gestione e per la disposizione in memoria degli indirizzi **Seg:Offset**; in base a questa convenzione, la componente **Offset** deve sempre precedere la componente **Seg**.
Supponiamo, ad esempio, di voler disporre in una locazione di memoria da **32** bit, un indirizzo **FAR** formato dalla coppia **0CFEh:04B8h**, cioe', **Seg=0CFEh**, **Offset=04B8h**; se la locazione di

memoria si trova all'offset **0004h** di un segmento di programma, allora, l'indirizzo **FAR** deve essere disposto in questo modo:

Contenuto	0Ch	FEh	04h	B8h
Offset	0007h	0006h	0005h	0004h

Come si puo' notare, la componente **Offset=04B8h** si trova all'offset **0004h**, mentre la componente **Seg=0CFEh** si trova all'offset **0006h**, cioe', **16** bit piu' avanti.

La situazione e' analoga nel caso in cui la componente **Offset** sia formata da **32** bit; volendo disporre, ad esempio, l'indirizzo **01FBh:03C8F284h**, in una locazione di memoria da **48** bit che si trova all'offset **0004h** di un segmento di programma, otteniamo la seguente situazione:

Contenuto	01h	FBh	03h	C8h	F2h	84h
Offset	0009h	0008h	0007h	0006h	0005h	0004h

In questo caso, si nota che la componente **Offset=03C8F284h** si trova all'offset **0004h**, mentre la componente **Seg=01FBh** si trova all'offset **0008h**, cioe', **32** bit piu' avanti.

Il programmatore e' tenuto ad osservare rigorosamente questa convenzione; in caso contrario, si ottengono programmi che, in fase di esecuzione, vanno sicuramente in crash!

Tutte le istruzioni della **CPU** che operano sugli indirizzi **FAR**, presuppongono che la struttura di questi indirizzi, segua la convenzione appena enunciata; tra queste particolari istruzioni, troviamo quelle rappresentate dai mnemonici **LDS**, **LES**, **LFS**, **LGS** e **LSS**.
Questi mnemonici, indicano la generica istruzione **Load Far Pointer** (caricamento di un indirizzo **FAR Seg:Offset**, in una coppia **SegReg:Reg**); le uniche forme consentite per queste istruzioni (indicate con il simbolo generico **LXS**), sono le seguenti:

```
LXS   Reg16, Mem16:Mem16
LXS   Reg32, Mem16:Mem32
```

Possiamo dire quindi che, se l'operando **DEST** e' un **Reg16**, allora l'operando **SRC** deve essere una locazione di memoria da **32** bit contenente una coppia **Seg:Offset** da **16+16** bit; se l'operando **DEST** e' un **Reg32**, allora l'operando **SRC** deve essere una locazione di memoria da **48** bit contenente una coppia **Seg:Offset** da **16+32** bit.
La componente **Offset** di questa coppia, viene caricata nel **Reg** che rappresenta l'operando **DEST**; la componente **Seg** di questa coppia, viene caricata nel **SegReg** specificato dal mnemonico dell'istruzione stessa (**DS**, **ES**, **FS**, **GS** o **SS**).

Nota importante.
Con una istruzione del tipo:

```
lss sp, Mem16:Mem16
```

e' possibile inizializzare rapidamente la coppia **SS:SP**; in questo caso, non bisogna preoccuparsi del possibile arrivo di una richiesta di interruzione. Infatti, durante la fase di esecuzione di una qualsiasi istruzione, tutte le interruzioni vengono, ovviamente, interdette; solo dopo l'elaborazione dell'istruzione, la **CPU** provvede a rispondere ad eventuali richieste di interruzione.

In relazione alle istruzioni **LDS**, **LES**, **LFS**, **LGS** e **LSS**, si possono presentare due casi fondamentali, che vengono analizzati nel seguito.

16.8.1 Operando DEST di tipo Reg16

Nel segmento dati (ad esempio, **DATASEGM**) del nostro programma, definiamo i seguenti dati statici:

```
FarPointer  dd    0
VarString   db    'Assembly Programming', 0
```

Supponiamo ora di voler stampare la stringa **VarString**, con la procedura **writeString** della libreria **EXELIB**; come sappiamo, questa procedura richiede che **ES:DI** punti alla stringa **C** da stampare, e che **DX** contenga le coordinate di output. In presenza di una direttiva **ASSUME** che associa **DATASEGM** a **DS**, possiamo scrivere allora le seguenti istruzioni:

```
mov      ax, offset VarString       ; ax = Offset VarString
mov      word ptr FarPointer[0], ax ; salva la componente Offset
mov      ax, seg VarString          ; ax = Seg VarString
mov      word ptr FarPointer[2], ax ; salva la componente Seg
les      di, FarPointer             ; ES:DI punta a VarString
mov      dx, 0400h                  ; riga 4, colonna 0
call     writeString                ; mostra la stringa
```

Dai manuali della **CPU**, si ricava che il codice macchina dell'istruzione:
`les di, FarPointer`
e' formato dal campo **Opcode 11000100b** (**C4h**), seguito dal campo **mod_reg_r/m** e da un **Disp16** (cioe', dall'offset di **FarPointer**); abbiamo quindi:
`Opcode = 11000100b = C4h`
Il registro **DEST** e' **DI**, per cui **reg=111b**; l'operando **SRC** e' un **Disp16**, per cui **mod=00b**, **r/m=110b**. Otteniamo quindi:
`mod_reg_r/m = 00111110b = 3Eh`
Supponendo che **FarPointer** si trovi all'offset **000Ah** di **DATASEGM**, otteniamo:
`Disp16 = 0000000000001010b = 000Ah`
L'assembler genera quindi il codice macchina:
`11000100b 00111110b 0000000000001010b = C4h 3Eh 000Ah`
Come possiamo notare, nel codice macchina non e' presente nessun prefisso **66h** (operandi a **32** bit) o **67h** (indirizzi a **32** bit); infatti, siccome l'operando **DEST** e' di tipo **SegReg:Reg16**, la **CPU** tratta l'operando **SRC** come **Mem16:Mem16**.
Quando la **CPU** incontra questo codice macchina, trasferisce in **DI** i **16** bit contenuti nella locazione di memoria **DS:000Ah**, e in **ES** i **16** bit contenuti nella locazione di memoria **DS:000Ch**.

Affinche' questo esempio funzioni in modo corretto, e' fondamentale che l'indirizzo **Seg:Offset** di **VarString**, venga disposto in **FarPointer** nel rispetto della convenzione enunciata in precedenza; la componente **Offset** di **VarString** deve essere posta nei **16** bit meno significativi di **FarPointer**, mentre la componente **Seg** di **VarString** deve essere posta nei **16** bit piu' significativi di **FarPointer**.

Se vogliamo verificare in pratica la corretta disposizione in **FarPointer**, dell'indirizzo **Seg:Offset** di **VarString**, possiamo scrivere il seguente codice (che deve essere aggiunto a quello del precedente esempio):

```
mov      ax, offset VarString       ; ax = Offset VarString
mov      dx, 0600h                  ; riga 6, colonna 0
call     writeHex16                 ; mostra l'Offset di VarString
mov      ax, seg VarString          ; ax = Seg VarString
mov      dx, 0700h                  ; riga 7, colonna 0
call     writeHex16                 ; mostra il Seg di VarString
mov      eax, FarPointer            ; eax = Seg:Offset di VarString
mov      dx, 0800h                  ; riga 8, colonna 0
call     writeHex32                 ; mostra l'indirizzo di VarString
```

16.8.2 Operando DEST di tipo Reg32

Nel segmento dati (ad esempio, **DATASEGM**) del nostro programma, definiamo i seguenti dati statici:

```
Seg16          dw    1CF8h          ; componente Seg a 16 bit
Off32          dd    3AC8F69Bh      ; componente Offset a 32 bit
FarPointer48   df    0              ; puntatore FAR da 16+32 bit
```

Supponiamo ora di voler caricare la coppia **Seg16:Off32**, sia in **FarPointer48**, sia in **FS:ESI**; in presenza di una direttiva **ASSUME** che associa **DATASEGM** a **DS**, e nell'ipotesi che **DS=DATASEGM**, possiamo scrivere le seguenti istruzioni:

```
mov      bx, offset FarPointer48    ; ds:bx punta a FarPointer48
mov      eax, Off32                 ; eax = Off32
mov      dword ptr [bx+0], eax      ; salva Off32
mov      ax, Seg16                  ; ax = Seg16
mov      word ptr [bx+4], ax        ; salva Seg16
lfs      esi, [bx]                  ; FS:ESI = FarPointer48

; verifica

mov      ax, fs                     ; ax = Seg16
mov      dx, 0400h                  ; riga 4, colonna 0
call     writeHex16                 ; mostra 1CF8h
mov      eax, esi                   ; eax = Off32
mov      dx, 0500h                  ; riga 5, colonna 0
call     writeHex32                 ; mostra 3AC8F69Bh
```

Dai manuali della **CPU**, si ricava che il codice macchina dell'istruzione:
`lfs esi, [bx]`
e' formato dal campo **Opcode 00001111b 10110100b** (**0Fh B4h**), seguito dal campo **mod_reg_r/m**; abbiamo quindi:
`Opcode = 00001111b 10110100b = 0Fh B4h`
Il registro **DEST** e' **ESI**, per cui **reg=110b**; l'operando **SRC** e' un **effective address** del tipo **[BX]**, per cui **mod=00b**, **r/m=111b**. Otteniamo quindi:
`mod_reg_r/m = 00110111b = 37h`
L'operando **DEST** e' a **32** bit, per cui si rende necessario il prefisso **66h**; l'assembler genera quindi il codice macchina:
`01100110b 00001111b 10110100b 00110111b = 66h 0Fh B4h 37h`
Osserviamo che in relazione all'operando **[BX]**, non e' necessario l'operatore **FWORD PTR**; infatti, siccome l'operando **DEST** e' di tipo **SegReg:Reg32**, la **CPU** tratta l'operando **SRC** come

Mem16:Mem32.
Quando la **CPU** incontra questo codice macchina, trasferisce in **ESI** i **32** bit contenuti nella locazione di memoria **DS:(BX+0)**, e in **FS** i **16** bit contenuti nella locazione di memoria **DS:(BX+4)**.

16.8.3 Effetti provocati da LDS, LES, LFS, LGS e LSS, sugli operandi e sui flags

L'esecuzione delle istruzioni **LDS**, **LES**, **LFS**, **LGS** e **LSS**, modifica il contenuto del solo operando **DEST**, che viene sovrascritto dal contenuto di **SRC**; il contenuto dell'operando **SRC** rimane inalterato.
Anche per queste istruzioni, bisogna prestare attenzione ai casi del tipo:
```
les bx, [bx+si+0004h]
```
Dopo l'esecuzione di questa istruzione, il contenuto della locazione di memoria puntata da **DS:(BX+SI+0004h)** rimane inalterato; invece, il contenuto del registro puntatore **BX** viene modificato.

L'esecuzione delle istruzioni **LDS**, **LES**, **LFS**, **LGS** e **LSS**, non modifica nessuno dei campi presenti nel **Flags Register**.

16.9 Le istruzioni IN e OUT

Nei precedenti capitoli, e' stato detto che ciascuna periferica collegata al computer, comunica con la **CPU** attraverso proprie aree di memoria che vengono chiamate **porte hardware**; si usa il termine "porta" proprio perche', grazie ad essa, i dati scambiati con la **CPU** possono entrare nella periferica, o uscire dalla periferica.
Ogni periferica puo' essere dotata di una sola porta hardware, come nel caso dei joystick, o di numerosissime porte hardware, come nel caso delle schede video; considerando proprio il caso della scheda video, si puo' dire che questa periferica, oltre ad essere dotata di numerose porte hardware propriamente dette, presenta anche un'area riservata alla memoria video (**VRAM**), formata spesso da milioni di byte, che formalmente possiamo equiparare a milioni di porte hardware.
Quando si deve comunicare con una memoria periferica molto grande (come nel caso della **VRAM**), si ricorre ad un metodo chiamato **I/O Memory Mapped** (input/output mappato in **RAM**); in pratica, la memoria periferica viene "mappata" in un'area della memoria centrale del computer, chiamata **finestra**, in modo che tutte le operazioni di **I/O** compiute su questa finestra, si ripercuotano sulla memoria periferica stessa.
Tutte le porte hardware che fanno capo, invece, ad aree di memoria molto piccole, vengono individuate assegnando a ciascuna di esse un indirizzo fisico, proprio come accade per i vari byte che formano la **RAM**; quando la **CPU** vuole comunicare con una di queste porte hardware, deve caricare il relativo indirizzo sull'**Address Bus**. Attraverso la logica di controllo, la **CPU** indica che quell'indirizzo si riferisce proprio ad una porta hardware, e non ad una locazione della **RAM** (vedere, ad esempio, la Figura 9 del Capitolo 7); dopo aver svolto queste operazioni, la **CPU** e' pronta per comunicare con la porta hardware attraverso il **Data Bus**.

Tutte le **CPU** precedenti l'**80386**, utilizzano gli **8** bit meno significativi dell'**Address Bus**, per specificare gli indirizzi delle varie porte hardware; queste **CPU** quindi, sono in grado di gestire sino a 2^8**=256** porte hardware differenti.
Le **CPU 80386** e superiori, utilizzano invece i **16** bit meno significativi dell'**Address Bus**; in questo modo, possono gestire sino a 2^{16}**=65536** porte hardware differenti.

Nota importante.
Nei **SO** piu' "grossolani", come il **DOS** e **Windows 95/98** (che a loro volta si appoggiano sul **DOS**), l'unico utente presente puo' accedere in modo diretto a tutte le porte hardware del **PC**; nel caso, invece, dei **SO** piu' professionali, come **Windows 2000/XP**, **Linux**, **UNIX**, etc, per poter accedere in modo diretto alle porte hardware del **PC**, e' necessario richiedere appositi permessi che, in genere, vengono concessi solo al cosiddetto **amministratore di sistema** (utente **root** per chi usa **Linux/UNIX**).

Le istruzioni che la **CPU** mette a disposizione per le comunicazioni con le porte hardware, sono rappresentate dai due mnemonici **IN** e **OUT**; l'istruzione **IN** serve per leggere dati da una porta hardware, mentre l'istruzione **OUT** serve per scrivere dati in una porta hardware.
L'utilizzo pratico di queste istruzioni, viene trattato in dettaglio nella sezione **Assembly Avanzato**; si raccomanda vivamente di non utilizzare **IN** e **OUT** per effettuare letture/scritture a caso nelle porte hardware, in quanto si potrebbero verificare malfunzionamenti del **PC**.

16.9.1 L'istruzione IN

Con il mnemonico **IN**, si indica l'istruzione **INput from port**; (lettura dati da una porta hardware); le uniche forme lecite per questa istruzione, sono le seguenti:

```
IN    AL, Imm8
IN    AX, Imm8
IN    EAX, Imm8
IN    AL, DX
IN    AX, DX
IN    EAX, DX
```

L'indirizzo della porta hardware a cui accedere in lettura (operando **SRC**), puo' essere espresso attraverso un **Imm8**, oppure attraverso il registro **DX**; con un **Imm8** possiamo specificare una tra **256** possibili porte, mentre con **DX** possiamo specificare una tra **65536** possibili porte (se si utilizza **DX** per specificare un indirizzo a **8** bit, e' importante che gli **8** bit piu' significativi dello stesso **DX**, vengano posti a zero).
L'operando **DEST**, deve essere obbligatoriamente l'accumulatore; l'ampiezza in bit (**8/16/32**) dell'accumulatore (**AL/AX/EAX**), determina il numero di byte (**1/2/4**) da leggere dalla porta.

Nel Capitolo 14, sono stati illustrati alcuni semplici esempi relativi all'accesso alle porte hardware; tali esempi, funzionano solo con i **SO** che permettono l'accesso diretto alle porte hardware.

16.9.2 L'istruzione OUT

Con il mnemonico **OUT**, si indica l'istruzione **OUTput to port**; (scrittura dati in una porta hardware); le uniche forme lecite per questa istruzione, sono le seguenti:

```
OUT   Imm8, AL
OUT   Imm8, AX
OUT   Imm8, EAX
OUT   DX, AL
OUT   DX, AX
OUT   DX, EAX
```

L'operando **SRC**, deve essere obbligatoriamente l'accumulatore; l'ampiezza in bit (**8/16/32**) dell'accumulatore (**AL/AX/EAX**), determina il numero di byte (**1/2/4**) da scrivere nella porta.

L'indirizzo della porta hardware a cui accedere in scrittura (operando **DEST**), puo' essere espresso attraverso un **Imm8**, oppure attraverso il registro **DX**; con un **Imm8** possiamo specificare una tra **256** possibili porte, mentre con **DX** possiamo specificare una tra **65536** possibili porte (se si utilizza **DX** per specificare un indirizzo a **8** bit, e' importante che gli **8** bit piu' significativi dello stesso **DX**, vengano posti a zero).
Osserviamo che per questa particolare istruzione, l'operando **DEST** puo' essere di tipo **Imm**; ovviamente, in questo caso l'**Imm** rappresenta l'indirizzo di una porta hardware, e non una locazione in cui scrivere dati.

16.9.3 Effetti provocati da IN e OUT sugli operandi e sui flags

L'esecuzione dell'istruzione **IN**, modifica il contenuto del solo accumulatore; il contenuto della porta da cui si effettua la lettura, rimane inalterato.
L'esecuzione dell'istruzione **OUT**, modifica il contenuto della porta in cui si effettua la scrittura; il contenuto dell'accumulatore, rimane inalterato.

L'esecuzione delle istruzioni **IN** e **OUT**, non modifica nessuno dei campi presenti nel **Flags Register**.

16.10 L'istruzione XCHG

Un caso classico che si presenta nella programmazione, consiste nella eventualita' di dover scambiare il contenuto di due variabili; vediamo un esempio pratico riferito al linguaggio **Pascal**.
Supponiamo di aver definito una variabile **IntVar1=12850**, e una variabile **IntVar2=27500**, entrambe di tipo **Integer** (intero con segno a **16** bit); per effettuare lo scambio del contenuto di **IntVar1** e **IntVar2**, si definisce una terza variabile, ad esempio, **Temp**, sempre di tipo **Integer**, e si scrivono le tre classiche istruzioni:

```
Temp:= IntVar1;       { Temp = 12850 }
IntVar1:= IntVar2;    { IntVar1 = 27500 }
IntVar2:= Temp;       { IntVar2 = 12850}
```

Dopo l'esecuzione di queste tre istruzioni, si ottiene **IntVar1=27500**, e **IntVar2=12850**; si e' verificato quindi lo scambio del contenuto delle due variabili.
Il codice utilizzato, appare molto compatto grazie al fatto che, i linguaggi di alto livello, permettono di simulare, via software, un trasferimento dati tra due locazioni di memoria; e' chiaro pero' che al momento di convertire il programma in codice macchina, il compilatore **Pascal** deve tenere conto del fatto che la **CPU** proibisce questo tipo di operazione. Di conseguenza, il compilatore stesso deve trovare le istruzioni **Assembly** che permettano di scambiare il contenuto di **IntVar1** e **IntVar2**, nel modo piu' rapido e piu' compatto possibile; una soluzione potrebbe essere la seguente:

```
push     IntVar1     ; FFh 36h 0000h - salva 12850 nello stack
push     IntVar2     ; FFh 36h 0002h - salva 27500 nello stack
pop      IntVar1     ; 8Fh 06h 0000h - copia 27500 in IntVar1
pop      IntVar2     ; 8Fh 06h 0002h - copia 12850 in IntVar2
```

Questo primo metodo genera un codice macchina da **16** byte, e con una **CPU 80486** richiede circa **4+4+6+6=20** cicli macchina; un'altra soluzione potrebbe essere la seguente:

```
mov      ax, IntVar1    ; A1h 0000h     - ax = 12850
mov      bx, IntVar2    ; 8Bh 1Eh 0002h - bx = 27500
mov      IntVar1, bx    ; 89h 1Eh 0000h - IntVar1 = 27500
mov      IntVar2, ax    ; A3h 0002h     - IntVar2 = 12850
```

Questo secondo metodo genera un codice macchina da **12** byte, e con una **CPU 80486** richiede circa **1+1+1+1=4** cicli macchina; come si puo' notare, questo metodo e' piu' compatto, e circa **5** volte piu' veloce di quello precedente!

Se siamo interessati, non solo alla velocita' del codice, ma anche alla sua compattezza, possiamo servirci di una apposita istruzione, rappresentata dal mnemonico **XCHG**; questo mnemonico indica l'istruzione **eXCHanGe register/memory with register** (scambia il contenuto di un **Reg/Mem** e di un **Reg**).
Le uniche forme lecite per questa istruzione, sono le seguenti:

```
XCHG     Reg, Reg
XCHG     Reg, Mem
XCHG     Mem, Reg
```

Trattandosi di un trasferimento dati "incrociato", i due operandi devono avere la stessa ampiezza in bit (che puo' essere **8**, **16** o **32**); e' proibito l'impiego di operandi di tipo **SegReg** o **Imm**.

Ripetiamo l'esempio precedente, servendoci dell'istruzione **XCHG**; in questo caso, possiamo scrivere:

```
mov      ax, IntVar1    ; A1h 0000h     - ax = 12850
xchg     IntVar2, ax    ; 87h 06h 0002h - IntVar2 = 12850, ax = 27500
mov      IntVar1, ax    ; A3h 0000h     - IntVar1 = 27500
```

Questo terzo metodo genera un codice macchina da **10** byte, e con una **CPU 80486** richiede circa **1+5+1=7** cicli macchina; come si puo' notare, questo metodo presenta una estrema compattezza, ed e' anche piuttosto veloce. Come si puo' facilmente intuire, nel caso di un programma che contiene numerosissimi scambi tra variabili, l'uso di **XCHG** porta ad una notevole riduzione delle dimensioni del codice.

16.10.1 Uso di XCHG con operandi di ampiezza arbitraria

Con le conoscenze che abbiamo acquisito, possiamo facilmente utilizzare le istruzioni illustrate in questo capitolo, con operandi di ampiezza arbitraria; in sostanza, possiamo effettuare trasferimenti di dati, anche tra operandi di ampiezza superiore a **32** bit.
Come esempio generale, vediamo una applicazione dell'istruzione **XCHG** per lo scambio di due **QWORD**; a tale proposito, nel blocco dati (ad esempio, **DATASEGM**) del nostro programma, inseriamo le seguenti definizioni:

```
VarXchg1    dq    3CF2A1B87951F2E6h
VarXchg2    dq    69E1CC28C6AB1189h
```

Se abbiamo a disposizione una **CPU 80386** o superiore, non dobbiamo fare altro che operare via hardware, sulle singole **DWORD** dei dati di tipo **QWORD**; in questo modo, sfruttiamo al massimo le prestazioni della **CPU**.

In presenza di una direttiva **ASSUME** che associa **DATASEGM** a **DS**, possiamo applicare il seguente metodo:

```
mov      eax, dword ptr VarXchg1[0] ; eax = prima DWORD di VarXchg1
xchg     dword ptr VarXchg2[0], eax ; prima parte dello scambio
mov      dword ptr VarXchg1[0], eax ; completa lo scambio

mov      eax, dword ptr VarXchg1[4] ; eax = seconda DWORD di VarXchg1
xchg     dword ptr VarXchg2[4], eax ; prima parte dello scambio
mov      dword ptr VarXchg1[4], eax ; completa lo scambio

; per visualizzare una QWORD con writeHex32, possiamo
; stampare due DWORD affiancate

mov      eax, dword ptr VarXchg1[4] ; DWORD alta di VarXchg1
mov      dx, 0400h                  ; riga 4, colonna 0
call     writeHex32                 ; stampa la DWORD
mov      eax, dword ptr VarXchg1[0] ; DWORD bassa di VarXchg
mov      dx, 0408h                  ; riga 4, colonna 8
call     writeHex32                 ; stampa la DWORD

mov      eax, dword ptr VarXchg2[4] ; DWORD alta di VarXchg2
mov      dx, 0500h                  ; riga 5, colonna 0
call     writeHex32                 ; stampa la DWORD
mov      eax, dword ptr VarXchg2[0] ; DWORD bassa di VarXchg2
mov      dx, 0508h                  ; riga 5, colonna 8
call     writeHex32                 ; stampa la DWORD
```

L'operatore **DWORD PTR** e' necessario in quanto abbiamo definito **VarXchg1** e **VarXchg2** come variabili di tipo **QWORD**; osserviamo inoltre che per visualizzare in modo corretto un dato di tipo **QWORD** con **writeHex32**, dobbiamo iniziare dalla **DWORD** piu' significativa (in caso contrario, **writeHex32** stampa delle **H** sul numero da visualizzare).

Nel caso in cui le variabili abbiano una ampiezza pari a decine e decine di byte, il metodo da applicare e' del tutto simile a quello appena illustrato; a tale proposito, supponiamo di voler scambiare il contenuto delle seguenti due variabili:
`BigVar1 = 99DD47B194B611F316F942EC3AC8F2BBh`
e:
`BigVar2 = 3B00C8C2668AB6A93CE1941D8F66B147h`
Se vogliamo disporre in memoria queste due variabili, nel rispetto della convenzione **little-endian**, dobbiamo scrivere le seguenti definizioni:

```
BigVar1     dd    3AC8F2BBh, 16F942ECh, 94B611F3h, 99DD47B1h
BigVar2     dd    8F66B147h, 3CE1941Dh, 668AB6A9h, 3B00C8C2h
```

Come si puo' notare, le varie **DWORD** di ogni variabile vengono disposte, una di seguito all'altra, a partire da quella meno significativa; in questo modo, l'assembler legge le **DWORD** a partire da quella piu' a sinistra, e le dispone in memoria ad indirizzi via via crescenti.
Alternativamente, si puo' anche scrivere:

```
BigVar1     dq    16F942EC3AC8F2BBh, 99DD47B194B611F3h
BigVar2     dq    3CE1941D8F66B147h, 3B00C8C2668AB6A9h
```

Anche in questo caso, le varie **QWORD** devono essere disposte, una di seguito all'altra, a partire da quella meno significativa.

A questo punto, tutto quello che dobbiamo fare, consiste nel gestire **BigVar1** e **BigVar2**, suddividendole in tante **DWORD**; servendoci di due registri puntatori, e nell'ipotesi che **DS=DATASEGM**, possiamo scrivere:

```
mov      si, offset BigVar1   ; ds:si punta a BigVar1
mov      di, offset BigVar2   ; ds:di punta a BigVar2
```

In questo modo, le quattro **DWORD** di **BigVar1** sono rappresentate da **[SI+0]**, **[SI+4]**, **[SI+8]** e **[SI+12]**, mentre le quattro **DWORD** di **BigVar2** sono rappresentate da **[DI+0]**, **[DI+4]**, **[DI+8]** e **[DI+12]**; per scambiare, ad esempio le **DWORD** meno significative di **BigVar1** e **BigVar2**, possiamo scrivere:

```
mov      eax, [si+0]     ; eax = prima DWORD di BigVar1
xchg     [di+0], eax     ; prima parte dello scambio
mov      [si+0], eax     ; completa lo scambio
```

Grazie alla presenza del registro **EAX**, non e' necessario l'uso dell'operatore **DWORD PTR**.
Al momento di visualizzare, con **writeHex32**, le due variabili **BigVar1** e **BigVar2**, dobbiamo ricordarci di partire con le rispettive **DWORD** piu' significative, e cioe' **[SI+12]** e **[DI+12]**.

In un prossimo capitolo, verranno illustrate le istruzioni di salto; con tali istruzioni, potremo realizzare delle iterazioni (loop), che ci permetteranno di rendere estremamente compatto il codice presentato nei precedenti esempi.

16.10.2 Effetti provocati da XCHG sugli operandi e sui flags

L'esecuzione dell'istruzione **XCHG**, modifica il contenuto di entrambi gli operandi; inoltre, bisogna prestare attenzione ai soliti casi del tipo:

```
xchg bx, [bx+di+002Ah]
```

Dopo l'esecuzione di questa istruzione, il contenuto del registro puntatore **BX** viene modificato.

Analogamente a quanto abbiamo visto per le istruzioni del tipo:

```
mov cx, cx
```

anche le istruzioni del tipo:

```
xchg dx, dx
```

non provocano nessuna modifica del contenuto dell'operando registro impiegato, contemporaneamente, come sorgente e come destinazione; proprio per questo motivo, gli assembler possono utilizzare queste particolari istruzioni, in sostituzione di **NOP**, per inserire dei buchi di memoria all'interno dei segmenti di codice.
In particolare, l'istruzione:

```
xchg ax, ax
```

viene considerata equivalente ad una istruzione **NOP**; infatti, in entrambi i casi viene generato il codice macchina **90h**.

L'esecuzione dell'istruzione **XCHG**, non modifica nessuno dei campi presenti nel **Flags Register**.

Capitolo 17 - Istruzioni aritmetiche

In questo capitolo vengono illustrate le principali istruzioni aritmetiche messe a disposizione dalla famiglia delle **CPU 80x86**; queste istruzioni permettono di effettuare svariate operazioni come, addizioni, sottrazioni, moltiplicazioni, divisioni, comparazioni numeriche, conversioni numeriche, cambiamenti di segno, etc.
Come e' stato spiegato nel capitolo sulla matematica del computer, una **CPU** con architettura a **n** bit gestisce via hardware qualsiasi operazione che coinvolge numeri interi aventi una ampiezza massima di **n** bit; tutte le operazioni su numeri interi aventi una ampiezza in bit superiore a **n**, devono essere gestite via software dal programmatore. Nel seguito del capitolo, vengono mostrati semplici esempi pratici che illustrano il procedimento da seguire per operare su numeri di grosse dimensioni; questi esempi, verranno poi ampliati e ottimizzati nei capitoli successivi.
Tutte le istruzioni aritmetiche, lavorano su operandi di tipo **Reg**, **Mem** e **Imm**; per ovvi motivi, e' proibito l'utilizzo di operandi di tipo **SegReg**.
Le istruzioni aritmetiche relative ai numeri in formato **BCD** o **Binary Coded Decimal** (decimale codificato in binario), verranno illustrate nel prossimo capitolo.

17.1 L'istruzione ADD

Con il mnemonico **ADD** si indica l'istruzione **ADDition** (addizione tra **SRC** e **DEST**); incontrando questa istruzione, la **CPU** somma il contenuto di **SRC** con il contenuto di **DEST**, e mette il risultato nell'operando **DEST**.
Entrambi gli operandi devono essere specificati esplicitamente dal programmatore, e devono avere la stessa ampiezza in bit; le uniche forme lecite per l'istruzione **ADD**, sono le seguenti:

```
ADD      Reg, Reg
ADD      Reg, Mem
ADD      Mem, Reg
ADD      Reg, Imm
ADD      Mem, Imm
```

Per poter analizzare degli esempi pratici, consideriamo le seguenti definizioni presenti in un blocco dati chiamato **DATASEGM**:

```
VarByte  db    210          ; intero a 8 bit
VarWord  dw    12890        ; intero a 16 bit
VarDword dd    3896580      ; intero a 32 bit
```

Ovviamente, agli operandi di tipo **Mem** si applicano tutti i concetti esposti nel precedente capitolo in relazione agli indirizzamenti; salvo casi particolari, tali concetti non verranno piu' ripetuti.

Supponiamo di avere **BX=14980** e **DX=20800**, e consideriamo la seguente istruzione:
```
add bx, dx
```
Quando la **CPU** incontra questa istruzione, calcola:
```
14980 + 20800 = 35780
```
Il risultato **35780**, viene poi sistemato nei **16** bit del registro **BX**.

Supponiamo di avere **CH=38**, e consideriamo l'istruzione:
```
add ch, VarByte
```
Quando la **CPU** incontra questa istruzione, calcola:
```
38 + 210 = 248
```
Il risultato **248**, viene poi sistemato negli **8** bit del registro **CH**.

Se **SI=20000**, e **DS:(BX+DI+0002h)** punta a **VarWord**, possiamo scrivere l'istruzione:
```
add [bx+di+0002h], si
```
La presenza del registro **SI** indica all'assembler che gli operandi sono a **16** bit; quando la **CPU** incontra questa istruzione, calcola:
```
12890 + 20000 = 32890
```
Il risultato **32890**, viene poi sistemato nella locazione di memoria (**VarWord**) che si trova all'indirizzo **DS:(BX+DI+0002h)**.

In presenza della dichiarazione:
```
PRESSIONE = 450000
```
e con **ES:DI** che punta a **VarDword**, possiamo scrivere la seguente istruzione:
```
add dword ptr es:[di], PRESSIONE
```
L'operatore **DWORD PTR** e' necessario per indicare all'assembler che gli operandi sono a **32** bit; il segment override e' necessario perche' **ES** non e' il registro di segmento naturale per i dati.
Quando la **CPU** incontra questa istruzione, calcola:
```
3896580 + 450000 = 4346580
```
Il risultato **4346580**, viene poi sistemato nella locazione di memoria (**VarDword**) che si trova all'indirizzo **ES:DI**.

17.1.1 Istruzione ADD con operando sorgente di tipo Imm

Nel campo **Opcode** del codice macchina di una qualsiasi istruzione aritmetica che coinvolge un eventuale operando sorgente di tipo **Imm**, e' sempre presente il bit **s** o **sign bit**; come sappiamo, questo bit indica alla **CPU** se e' necessario procedere all'estensione del bit di segno dello stesso **Imm**. In sostanza, se **s=0**, l'**Imm** non viene modificato dalla **CPU**; se, invece, **s=1**, l'**Imm** viene esteso attraverso il suo bit di segno, sino a raggiungere l'ampiezza in bit dell'operando **DEST**.
Se necessario, l'estensione dell'ampiezza dell'operando di tipo **Imm**, viene effettuata direttamente dall'assembler, che pone quindi, **s=0**; in alternativa, l'assembler puo' decidere di delegare questo compito alla **CPU**. In tal caso, l'assembler stesso pone **s=1**; considerando il fatto che questo aspetto vale per tutte le istruzioni aritmetiche, analizziamo alcuni esempi dettagliati.
E' importante ricordare che il programmatore, ha il compito di assegnare ad un **Imm**, un valore compatibile con l'ampiezza in bit degli operandi delle istruzioni che coinvolgono lo stesso **Imm**; in caso contrario, si ottengono risultati privi di senso.

Dai manuali della **CPU**, si ricava che il codice macchina generico per una somma tra sorgente **Imm** e destinazione **Reg/Mem**, e' formato dall'**Opcode 100000sw**, seguito dal campo **mod_000_r/m** e da un **Imm**.

Supponiamo, ad esempio, di dichiarare un **Imm** che vogliamo trattare come intero senza segno a **8** bit; in tal caso, allo stesso **Imm** dobbiamo assegnare un valore compreso tra **0** e **255**.
Come esempio pratico, consideriamo la seguente dichiarazione:
```
SUPERFICIE = 32
```
In assenza del segno esplicito, l'assembler tratta **SUPERFICIE** come un intero positivo (**+32**) che richiede almeno **8** bit; la rappresentazione binaria a **8** bit di **+32**, e' **00100000b** (**20h**).
Nell'ipotesi che **EBX=0171D86Eh** (**24238190**), consideriamo ora la seguente istruzione:
```
add ebx, SUPERFICIE
```
Questa istruzione deve produrre il risultato:
```
24238190 + 32 = 24238222
```
Gli operandi devono essere a **32** bit (**w=1**), per cui si rende necessaria l'estensione dell'ampiezza in bit di **SUPERFICIE**; l'assembler osserva che il bit di segno di **SUPERFICIE** e' **0**, compatibile quindi con il segno positivo. L'estensione dell'ampiezza di **SUPERFICIE** puo' essere delegata allora alla **CPU**; l'assembler pone **s=1**, per cui:

```
Opcode = 10000011b = 83h
```
Il registro destinazione e' **EBX**, per cui **r/m=011b**; per indicare che **r/m** codifica un registro, si pone, come sappiamo, **mod=11b**. Si ottiene quindi:
```
mod_000_r/m = 11000011b = C3h
```
L'operando sorgente e':
```
Imm = 00100000b = 20h
```
Gli operandi sono a **32** bit, per cui si rende necessario il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
01100110b 10000011b 11000011b 00100000b = 66h 83h C3h 20h
```
Quando la **CPU** incontra questa istruzione, estende a **32** bit il valore **00100000b** (attraverso il suo bit di segno **0**), e ottiene:
```
00000000000000000000000000100000b = 00000020h = +32
```
Successivamente, la **CPU** esegue la somma:
```
0171D86Eh + 00000020h = 0171D88Eh = 24238222
```
La **CPU** ha quindi eseguito correttamente la somma:
```
24238190 + 32 = 24238222
```
Il risultato **24238222** viene sistemato nel registro **EBX**; se vogliamo verificare in pratica l'esempio appena esposto, possiamo scrivere le seguenti istruzioni, che si servono della procedura **writeUdec32**, per la visualizzazione di un intero senza segno a **32** bit:

```
mov      ebx, 24238190     ; ebx = 24238190
add      ebx, SUPERFICIE   ; ebx = 24238190 + 32
mov      eax, ebx          ; eax = 24238222
mov      dx, 0400h         ; riga 4, colonna 0
call     writeUdec32       ; risultato in base 10
mov      dx, 0500h         ; riga 5, colonna 0
call     writeHex32        ; risultato in base 16
```

Supponiamo, ad esempio, di dichiarare un **Imm** che vogliamo trattare come intero con segno a **8** bit; in tal caso, allo stesso **Imm** dobbiamo assegnare un valore compreso tra **-128** e **+127**.
Come esempio pratico, consideriamo la seguente dichiarazione:
```
DISLIVELLO = -49
```
La rappresentazione binaria a **8** bit in complemento a **2** del numero negativo **-49**, e':
```
256 - 49 = 207 = 11001111b = CFh
```
Nell'ipotesi che **EBX=0171D86Eh** (**24238190**), consideriamo ora la seguente istruzione:
```
add ebx, DISLIVELLO
```
Questa istruzione deve produrre il risultato:
```
24238190 + (-49) = 24238141
```
Gli operandi devono essere a **32** bit (**w=1**), per cui si rende necessaria l'estensione dell'ampiezza in bit di **DISLIVELLO**; l'assembler osserva che il bit di segno di **DISLIVELLO** e' **1**, compatibile quindi con il segno negativo. L'estensione dell'ampiezza di **DISLIVELLO** puo' essere delegata allora alla **CPU**; l'assembler pone **s=1**, per cui:
```
Opcode = 10000011b = 83h
```
Il registro destinazione e' **EBX**, per cui **r/m=011b**; per indicare che **r/m** codifica un registro, si pone **mod=11b**. Si ottiene quindi:
```
mod_000_r/m = 11000011b = C3h
```
L'operando sorgente e':
```
Imm = 11001111b = CFh
```
Gli operandi sono a **32** bit, per cui si rende necessario il prefisso **66h**; l'assembler genera quindi il codice macchina:
```
01100110b 10000011b 11000011b 11001111b = 66h 83h C3h CFh
```
Quando la **CPU** incontra questa istruzione, estende a **32** bit il valore **11001111b** (attraverso il suo bit di segno **1**), e ottiene:
```
11111111111111111111111111001111b = FFFFFFCFh = -49
```

Successivamente, la **CPU** esegue la somma:
`0171D86Eh + FFFFFFCFh = 0171D83Dh = 24238141`
La **CPU** ha quindi eseguito correttamente la somma:
`24238190 + (-49) = 24238141`
Il risultato **24238141** viene sistemato nel registro **EBX**; se vogliamo verificare in pratica l'esempio appena esposto, possiamo scrivere le seguenti istruzioni, che si servono della procedura **writeSdec32**, per la visualizzazione di un intero con segno a **32** bit:

```
mov      ebx, 24238190      ; ebx = 24238190
add      ebx, DISLIVELLO    ; ebx = 24238190 + (-49)
mov      eax, ebx           ; eax = 24238141
mov      dx, 0400h          ; riga 4, colonna 0
call     writeSdec32        ; risultato in base 10
mov      dx, 0500h          ; riga 5, colonna 0
call     writeHex32         ; risultato in base 16
```

17.1.2 Effetti provocati da ADD sugli operandi e sui flags

L'esecuzione dell'istruzione **ADD**, modifica il contenuto del solo operando **DEST**, che viene sovrascritto dal risultato della somma appena effettuata; il contenuto dell'operando **SRC** rimane inalterato.
Anche per **ADD**, bisogna prestare attenzione ai soliti casi del tipo:
`add bx, [bx+di+002Ah]`
Dopo l'esecuzione di questa istruzione, il contenuto del registro puntatore **BX** viene modificato.

La possibilita' di effettuare una somma tra **Reg** e **Reg**, ci permette di scrivere istruzioni del tipo:
`add ax, ax`
Come si puo' facilmente constatare, l'esecuzione di questa istruzione fa' raddoppiare il contenuto originale di **AX**; se, ad esempio, **AX=2500**, dopo l'esecuzione di **ADD** si ottiene:
`AX = 2500 + 2500 = 5000`

L'esecuzione dell'istruzione **ADD**, e di qualsiasi altra istruzione aritmetica, modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** del **Flags Register**; come si puo' facilmente intuire, le modifiche apportate a questi flags assumono una importanza enorme, in quanto forniscono, sia alla **CPU**, sia al programmatore, informazioni dettagliate sul risultato scaturito dalla operazione appena eseguita.
Questi aspetti sono gia' stati esposti nel capitolo dedicato alla matematica del computer; vediamo alcuni esempi, che si riferiscono ad operandi a **16** bit, e che possono essere verificati attraverso le seguenti istruzioni:

```
mov      ax, valore1        ; ax = primo addendo
mov      bx, valore2        ; bx = secondo addendo
add      ax, bx             ; ax = ax + bx
push     ax                 ; preserva ax
call     showCPU            ; mostra AX, BX, FLAGS, etc.
pop      ax                 ; ripristina ax
mov      dx, 0400h          ; riga 4, colonna 0
call     writeSdec16        ; risultato in formato intero con segno
mov      dx, 0500h          ; riga 5, colonna 0
call     writeUdec16        ; risultato in formato intero senza segno
```

Come sappiamo, grazie all'aritmetica modulare, la **CPU** utilizza la sola istruzione **ADD** per operare, indifferentemente, su numeri interi con o senza segno; attraverso la consultazione degli opportuni flags, il programmatore ha la possibilita' di conoscere il risultato di una somma, sia nell'insieme degli interi con segno, sia nell'insieme degli interi senza segno.

Poniamo **AX=12850**, **BX=4700**, ed eseguiamo l'istruzione:
```
add ax, bx
```
In binario si ha:
```
AX = 12850 = 0011001000110010b
```
e:
```
BX = 4700 = 0001001001011100b
```
La **CPU** esegue quindi la somma:

```
0011001000110010b +
0001001001011100b =
-------------------
0100010010001110b     (17550)
```

Sulla base di questo risultato, la **CPU** pone:
```
CF = 0, PF = 1, AF = 0, ZF = 0, SF = 0, OF = 0
```
Per giustificare questa situazione, osserviamo innanzi tutto che il risultato e' diverso da **0**, per cui **ZF=0**; l'esecuzione della somma non ha prodotto nessun riporto dal primo nibble al secondo nibble, per cui **AF=0**. Negli **8** bit meno significativi del risultato, e' presente un numero pari (**4**) di **1**; di conseguenza, **PF=1**.
Nell'insieme degli interi senza segno, **0011001000110010b** rappresenta **12850**, mentre **0001001001011100b** rappresenta **4700**; la somma di questi due numeri, produce un risultato (**17550**) non superiore a **65535**, per cui **CF=0** (nessun riporto).
Nell'insieme degli interi con segno, **0011001000110010b** rappresenta **+12850**, mentre **0001001001011100b** rappresenta **+4700**; la somma di questi due numeri, produce un risultato (**+17550**)non superiore a **+32767**, per cui **OF=0**. Il bit piu' significativo del risultato e' **0**, per cui **SF=0** (numero positivo).

Poniamo **AX=12850**, **BX=-4700**, ed eseguiamo l'istruzione:
```
add ax, bx
```
In binario si ha:
```
AX = 12850 = 0011001000110010b
```
e:

BX = 2^{16} - 4700 = 60836 = 1110110110100100b

La **CPU** esegue quindi la somma:

```
0011001000110010b +
1110110110100100b =
-------------------
0001111111010110b     (8150)
```

Sulla base di questo risultato, la **CPU** pone:
```
CF = 1, PF = 0, AF = 0, ZF = 0, SF = 0, OF = 0
```
Per giustificare questa situazione, osserviamo innanzi tutto che il risultato e' diverso da **0**, per cui **ZF=0**; l'esecuzione della somma non ha prodotto nessun riporto dal primo nibble al secondo nibble, per cui **AF=0**. Negli **8** bit meno significativi del risultato, e' presente un numero dispari (**5**) di **1**; di conseguenza, **PF=0**.
Nell'insieme degli interi senza segno, **0011001000110010b** rappresenta **12850**, mentre **1110110110100100b** rappresenta **60836**; la somma di questi due numeri, produce un risultato (**73686**) superiore a **65535**, per cui **CF=1** (riporto). Osserviamo che in binario, **73686** si scrive **10001111111010110b**, cioe' **0001111111010110b** (**8150**) con riporto di **1**.

Nell'insieme degli interi con segno, **0011001000110010b** rappresenta **+12850**, mentre **1110110110100100b** rappresenta **-4700**; la somma di questi due numeri, produce un risultato (**+8150**) non superiore a **+32767**, per cui **OF=0**. Il bit piu' significativo del risultato e' **0**, per cui **SF=0** (numero positivo).

Poniamo **AX=-12812**, **BX=-18004**, ed eseguiamo l'istruzione:
```
add ax, bx
```
In binario si ha:
AX = 2^{16} - 12812 = 52724 = 1100110111110100b
e:
BX = 2^{16} - 18004 = 47532 = 1011100110101100b
La **CPU** esegue quindi la somma:

```
1100110111110100b +
1011100110101100b =
-------------------
1000011110100000b     (34720)
```

Sulla base di questo risultato, la **CPU** pone:
```
CF = 1, PF = 1, AF = 1, ZF = 0, SF = 1, OF = 0
```
Per giustificare questa situazione, osserviamo innanzi tutto che il risultato e' diverso da **0**, per cui **ZF=0**; l'esecuzione della somma ha prodotto un riporto dal primo nibble al secondo nibble, per cui **AF=1**. Negli **8** bit meno significativi del risultato, e' presente un numero pari (**2**) di **1**; di conseguenza, **PF=1**.
Nell'insieme degli interi senza segno, **1100110111110100b** rappresenta **52724**, mentre **1011100110101100b** rappresenta **47532**; la somma di questi due numeri, produce un risultato (**100256**) superiore a **65535**, per cui **CF=1** (riporto). Osserviamo che in binario, **100256** si scrive **11000011110100000b**, cioe' **1000011110100000b** (**34720**) con riporto di **1**.
Nell'insieme degli interi con segno, **1100110111110100b** rappresenta **-12812**, mentre **1011100110101100b** rappresenta **-18004**; la somma di questi due numeri, produce un risultato (**-30816**) non inferiore a
-32768, per cui **OF=0**. Il bit piu' significativo del risultato e' **1**, per cui **SF=1** (numero negativo); osserviamo che per i numeri con segno a **16** bit in complemento a **2**, **-30816** si scrive:
2^{16} - 30816 = 34720 = 1000011110100000b

Poniamo **AX=28500**, **BX=21370**, ed eseguiamo l'istruzione:
```
add ax, bx
```
In binario si ha:
AX = 28500 = 0110111101010100b
e:
BX = 21370 = 0101001101111010b
La **CPU** esegue quindi la somma:

```
0110111101010100b +
0101001101111010b =
-------------------
1100001011001110b     (49870)
```

Sulla base di questo risultato, la **CPU** pone:
```
CF = 0, PF = 0, AF = 0, ZF = 0, SF = 1, OF = 1
```

Per giustificare questa situazione, osserviamo innanzi tutto che il risultato e' diverso da **0**, per cui **ZF=0**; l'esecuzione della somma non ha prodotto nessun riporto dal primo nibble al secondo nibble, per cui **AF=0**. Negli **8** bit meno significativi del risultato, e' presente un numero dispari (**5**) di **1**; di conseguenza, **PF=0**.
Nell'insieme degli interi senza segno, **0110111101010100b** rappresenta **28500**, mentre **0101001101111010b** rappresenta **21370**; la somma di questi due numeri, produce un risultato (**49870**) non superiore a **65535**, per cui **CF=0** (nessun riporto).
Nell'insieme degli interi con segno, **0110111101010100b** rappresenta **+28500**, mentre **0101001101111010b** rappresenta **+21370**; la somma di questi due numeri, produce un risultato (**+49870**) superiore a **+32767**, per cui **OF=1**. Il bit piu' significativo del risultato e' **1**, per cui **SF=1** (numero negativo); infatti, per i numeri con segno a **16** bit in complemento a **2**, **49870** rappresenta:
$49870 - 2^{16} = -15666$
Sommando quindi due numeri positivi, abbiamo ottenuto un numero negativo; come sappiamo, la **CPU** si basa proprio su questo evento per individuare un overflow.
Nota importante.
Qualsiasi istruzione aritmetica eseguita su una porzione di un registro, non modifica in alcun modo il contenuto della parte restante del registro stesso; supponiamo, ad esempio, di avere **AX=3AFFh**, e consideriamo l'istruzione:

```
add al, 1
```

L'esecuzione di questa istruzione, produce il seguente risultato:

```
AL = FFh + 01h = 00h, con CF = 1
```

Il riporto che si e' verificato, viene notificato attraverso **CF**, e non si ripercuote in alcun modo, sul contenuto (**3Ah**) degli **8** bit piu' significativi (**AH**) di **AX**; in sostanza, dopo l'esecuzione di questa istruzione, otteniamo **AX=3A00h**.
Lo stesso discorso vale, ovviamente, per i **24** bit piu' significativi del registro **EAX**.

17.2 L'istruzione ADC

Con il mnemonico **ADC** si indica l'istruzione **ADd with Carry** (somma tra **SRC**, **DEST** e **CF**); incontrando questa istruzione, la **CPU** calcola:

```
DEST = DEST + SRC + CF
```

Gli operandi **SRC** e **DEST** devono essere specificati esplicitamente dal programmatore, e devono avere la stessa ampiezza in bit; il risultato della somma, viene disposto in **DEST**.
In analogia con **ADD**, le uniche forme lecite per l'istruzione **ADC**, sono le seguenti:

```
ADC      Reg, Reg
ADC      Reg, Mem
ADC      Mem, Reg
ADC      Reg, Imm
ADC      Mem, Imm
```

Come si puo' facilmente immaginare, l'istruzione **ADC** e' stata concepita, principalmente, per permettere di eseguire in modo semplice, somme tra operandi di ampiezza arbitraria; a tale proposito, si devono applicare tutti i concetti esposti nel capitolo sulla matematica del computer. In particolare, e' necessario ricordare che la somma:

```
DEST + SRC + CF
```

non puo' mai provocare due riporti consecutivi.
Tutte le considerazioni esposte per l'istruzione **ADD** in relazione agli effetti provocati sugli operandi e sui flags, restano valide anche per l'istruzione **ADC**.
Come accade per **ADD**, anche **ADC**, grazie all'aritmetica modulare, opera, indifferentemente, sui numeri interi con o senza segno; analizziamo il procedimento che bisogna seguire nei due casi che si possono presentare.

17.2.1 Somma tra numeri interi senza segno di ampiezza arbitraria

Supponiamo di voler eseguire la seguente somma tra interi senza segno a **96** bit:

```
3FB21866419CAF87A6B9E744h +
A394C276D79B3E4F69632FFFh =
---------------------------
E346DADD1937EDD7101D1743h
```

Procediamo innanzi tutto con la definizione delle due variabili; utilizzando, ad esempio, la direttiva **DD**, possiamo scrivere:

```
BigVar1  dd    0A6B9E744h, 419CAF87h, 3FB21866h
BigVar2  dd    69632FFFh, 0D79B3E4Fh, 0A394C276h
BigTotal dd    0, 0, 0
```

A questo punto possiamo procedere con l'esecuzione della somma; la somma tra le due **DWORD** meno significative viene effettuata con **ADD**, mentre le due somme successive, vengono effettuate con **ADC** in quanto bisogna tenere conto dell'eventuale riporto proveniente dalle somme precedenti. Il codice che esegue la somma assume il seguente aspetto:

```
mov      si, offset BigVar1       ; ds:si punta a BigVar1
mov      di, offset BigVar2       ; ds:di punta a BigVar2
mov      bx, offset BigTotal      ; ds:bx punta a BigTotal

; somma delle prime DWORD con ADD

mov      eax, [si+0]              ; eax = BigVar1[0]
add      eax, [di+0]              ; eax = BigVar1[0] + BigVar2[0]
mov      [bx+0], eax              ; BigTotal[0] = eax

; somma delle seconde DWORD con ADC

mov      eax, [si+4]              ; eax = BigVar1[4]
adc      eax, [di+4]              ; eax = BigVar1[4] + BigVar2[4] + CF
mov      [bx+4], eax              ; BigTotal[4] = eax

; somma delle terze DWORD con ADC

mov      eax, [si+8]              ; eax = BigVar1[8]
adc      eax, [di+8]              ; eax = BigVar1[8] + BigVar2[8] + CF
mov      [bx+8], eax              ; BigTotal[8] = eax

call     showCPU                  ; verifica dei flags
```

Dopo l'esecuzione dell'ultima somma (tra le due **DWORD** piu' significative), consultiamo **CF** per sapere se c'e' stato un riporto finale; nel nostro caso, **CF=0**, per cui il risultato e' valido cosi' com'e'.
Se vogliamo visualizzare il risultato con **writeHex32**, possiamo procedere nel solito modo; possiamo scrivere, ad esempio:

```
mov      bx, offset BigTotal      ; ds:bx punta a BigTotal

mov      eax, [bx+8]              ; eax = BigTotal[8]
mov      dx, 0400h                ; riga 4, colonna 0
call     writeHex32               ; visualizza eax

mov      eax, [bx+4]              ; eax = BigTotal[4]
mov      dx, 0408h                ; riga 4, colonna 8
call     writeHex32               ; visualizza eax

mov      eax, [bx+0]              ; eax = BigTotal[0]
mov      dx, 0410h                ; riga 4, colonna 16
call     writeHex32               ; visualizza eax
```

Come e' stato gia' spiegato nel capitolo precedente, per mostrare sullo schermo, con **writeHex32**, un numero esadecimale con ampiezza maggiore di **32** bit, e' necessario partire dalla **DWORD** piu' significativa; in caso contrario, **writeHex32** stamperebbe delle **H** sul numero da visualizzare.

17.2.2 Somma tra numeri interi con segno di ampiezza arbitraria

Naturalmente, l'esempio appena illustrato e' perfettamente valido anche quando i due addendi codificano numeri interi con segno a **96** bit; in tal caso, dobbiamo tenere presente che i numeri stessi risultano rappresentati in complemento a **2**, modulo $\mathbf{2^{96}}$. In base al concetto di **estensione del bit di segno**, possiamo dire che, tutti i numeri esadecimali a **96** bit, la cui cifra piu' significativa e' compresa tra **0** e **7** (cioe', tra **0000b** e **0111b**), sono positivi; tutti i numeri esadecimali a **96** bit, la cui cifra piu' significativa e' compresa tra **8** e **F** (cioe', tra **1000b** e **1111b**), sono negativi.

Per sommare tra loro numeri interi con segno di ampiezza arbitraria, si procede esattamente come mostrato nel precedente esempio; la prima somma viene eseguita con **ADD**, mentre le somme successive vengono eseguite con **ADC**. Dopo l'esecuzione dell'ultima somma (tra le due **DWORD** piu' significative), si deve consultare, non **CF**, ma bensi' **OF** e **SF**; e' chiaro, infatti, che il segno del risultato si trova codificato nel suo bit piu' significativo.
Se, dopo l'ultima somma, si ha **OF=0**, allora il risultato e' valido, e il relativo segno si trova codificato in **SF**; se, invece, dopo l'ultima somma, si ha **OF=1**, il risultato e' andato in overflow (in tal caso, il contenuto di **SF** non ha, ovviamente, nessun significato).

17.2.3 Effetti provocati da ADC sugli operandi e sui flags

L'esecuzione dell'istruzione **ADC**, modifica il contenuto del solo operando **DEST**, che viene sovrascritto dal risultato della somma appena effettuata; il contenuto dell'operando **SRC** rimane inalterato.
Anche per **ADC**, bisogna prestare attenzione ai soliti casi del tipo:
`adc bx, [bx+di+002Ah]`
Dopo l'esecuzione di questa istruzione, il contenuto del registro puntatore **BX** viene modificato.

L'esecuzione dell'istruzione **ADC**, modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** del **Flags Register**; il contenuto di questi campi, ha l'identico significato gia' illustrato per l'istruzione **ADD**.

17.3 L'istruzione SUB

Con il mnemonico **SUB** si indica l'istruzione **SUBtraction** (sottrazione tra **SRC** e **DEST**); incontrando questa istruzione, la **CPU** calcola:

```
DEST = DEST - SRC
```

Gli operandi **SRC** e **DEST** devono essere specificati esplicitamente dal programmatore, e devono avere la stessa ampiezza in bit; il risultato della sottrazione, viene disposto in **DEST**.
Le uniche forme lecite per l'istruzione **SUB**, sono le seguenti:

```
SUB      Reg, Reg
SUB      Reg, Mem
SUB      Mem, Reg
SUB      Reg, Imm
SUB      Mem, Imm
```

Appare evidente il fatto che **ADD** e **SUB**, sono istruzioni tra loro complementari; infatti, dati i due numeri interi **A** e **B**, possiamo scrivere:

```
A + B = A - (-B)
```

e:

```
A - B = A + (-B)
```

Non bisogna pero' dimenticare che nel caso di **SUB**, la **CPU** si serve del flag **CF** per segnalare, non un eventuale riporto, ma bensi' un eventuale prestito; analogamente, la **CPU** si serve del flag **AF**, per segnalare che nel corso della sottrazione, si e' verificato un eventuale prestito dal secondo nibble al primo nibble.
Tutti gli esempi presentati in precedenza per l'istruzione **ADD**, possono essere quindi ripetuti per l'istruzione **SUB**, sostituendo semplicemente il mnemonico **ADD** con il mnemonico **SUB**.

Grazie alla aritmetica modulare, la **CPU** puo' eseguire le sottrazioni senza dover distinguere tra interi con o senza segno; vediamo un esempio pratico.

Poniamo **AX=45200**, **BX=38260**, ed eseguiamo l'istruzione:

```
sub ax, bx
```

In binario si ha:

```
AX = 45200 = 1011000010010000b
```

e:

```
BX = 38260 = 1001010101110100b
```

La **CPU** esegue quindi la sottrazione:

```
1011000010010000b -
1001010101110100b =
-------------------
0001101100011100b     (6940)
```

Sulla base di questo risultato, la **CPU** pone:

```
CF = 0, PF = 0, AF = 1, ZF = 0, SF = 0, OF = 0
```

Per giustificare questa situazione, osserviamo innanzi tutto che il risultato e' diverso da **0**, per cui **ZF=0**; l'esecuzione della sottrazione ha prodotto un prestito dal secondo nibble al primo nibble, per cui **AF=1**. Negli **8** bit meno significativi del risultato, e' presente un numero dispari (**3**) di **1**; di conseguenza, **PF=0**.
Nell'insieme degli interi senza segno, **1011000010010000b** rappresenta **45200**, mentre **1001010101110100b** rappresenta **38260**; la sottrazione:

```
45200 - 38260 = 6940
```

produce un risultato positivo (minuendo maggiore del sottraendo), per cui **CF=0** (nessun prestito).

Nell'insieme degli interi con segno, **1011000010010000b** rappresenta:
`45200 - 2^16 = -20336`
mentre **1001010101110100b** rappresenta:
`38260 - 2^16 = -27276`
La sottrazione:
`-20336 - (-27276) = -20336 + 27276 = +6940`
produce un risultato non superiore a **+32767**, per cui **OF=0**; il bit piu' significativo del risultato e' **0**, per cui **SF=0** (numero positivo).

17.3.1 Effetti provocati da SUB sugli operandi e sui flags

L'esecuzione dell'istruzione **SUB**, modifica il contenuto del solo operando **DEST**, che viene sovrascritto dal risultato della sottrazione appena effettuata; il contenuto dell'operando **SRC** rimane inalterato.
Anche per **SUB**, bisogna prestare attenzione ai soliti casi del tipo:
```
sub di, [di]
```
Dopo l'esecuzione di questa istruzione, il contenuto del registro puntatore **DI** viene modificato.

Il metodo piu' semplice e intuitivo per azzerare il contenuto di un registro, consiste nel servirsi dell'istruzione **MOV**; se, ad esempio, vogliamo azzerare il contenuto del registro **ECX**, possiamo scrivere:
```
mov ecx, 0
```
La possibilita' di effettuare una sottrazione tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:
```
sub ecx, ecx
```
Anche in questo caso, l'effetto prodotto consiste nell'azzeramento del contenuto di **ECX**.

L'esecuzione dell'istruzione **SUB**, modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** del **Flags Register**; il contenuto di questi campi, ha lo stesso significato gia' illustrato per le istruzioni **ADD** e **ADC**.
Questa volta pero', **AF** indica se, nell'esecuzione della sottrazione, si e' verificato un eventuale prestito dal secondo nibble al primo nibble; analogamente, **CF** indica se la sottrazione si e' conclusa con un eventuale prestito (minuendo minore del sottraendo).

17.4 L'istruzione SBB

Con il mnemonico **SBB** si indica l'istruzione **SuBtraction with Borrow** (sottrazione tra **SRC**, **DEST** e **CF**); incontrando questa istruzione, la **CPU** calcola:
```
DEST = DEST - SRC - CF = DEST - (SRC + CF)
```
Gli operandi **SRC** e **DEST** devono essere specificati esplicitamente dal programmatore, e devono avere la stessa ampiezza in bit; il risultato della sottrazione, viene disposto in **DEST**.

In analogia con **SUB**, le uniche forme lecite per l'istruzione **SBB**, sono le seguenti:

```
SBB      Reg, Reg
SBB      Reg, Mem
SBB      Mem, Reg
SBB      Reg, Imm
SBB      Mem, Imm
```

Come si puo' facilmente immaginare, l'istruzione **SBB** e' stata concepita, principalmente, per permettere di eseguire in modo semplice, sottrazioni tra operandi di ampiezza arbitraria; a tale proposito, si devono applicare tutti i concetti esposti nel capitolo sulla matematica del computer. In particolare, e' necessario ricordare che la sottrazione:
```
DEST - SRC - CF
```
non puo' mai provocare due prestiti consecutivi.

Tutte le considerazioni esposte per l'istruzione **SUB** in relazione agli effetti provocati sugli operandi e sui flags, restano valide anche per l'istruzione **SBB**.

Come accade con **SUB**, anche **SBB**, grazie all'aritmetica modulare, opera, indifferentemente, sui numeri interi con o senza segno; analizziamo il procedimento che bisogna seguire nei due casi che si possono presentare.

17.4.1 Sottrazione tra numeri interi senza segno di ampiezza arbitraria

Supponiamo di voler eseguire la seguente sottrazione tra interi senza segno a **96** bit:

```
A394C276D79B3E4F69632FFFh -
3FB21866419CAF87A6B9E744h =
---------------------------
63E2AA1095FE8EC7C2A948BBh
```

Procediamo innanzi tutto con la definizione delle due variabili; utilizzando, ad esempio, la direttiva **DD**, possiamo scrivere:

```
BigVar1  dd    69632FFFh, 0D79B3E4Fh, 0A394C276h
BigVar2  dd    0A6B9E744h, 419CAF87h, 3FB21866h
BigTotal dd    0, 0, 0
```

A questo punto possiamo procedere con l'esecuzione della sottrazione; la sottrazione tra le due **DWORD** meno significative viene effettuata con **SUB**, mentre le due sottrazioni successive, vengono effettuate con **SBB** in quanto bisogna tenere conto dell'eventuale prestito richiesto dalle sottrazioni eseguite in precedenza.
Il codice che esegue la sottrazione assume il seguente aspetto:

```
mov      si, offset BigVar1      ; ds:si punta a BigVar1
mov      di, offset BigVar2      ; ds:di punta a BigVar2
mov      bx, offset BigTotal     ; ds:bx punta a BigTotal

; sottrazione delle prime DWORD con SUB

mov      eax, [si+0]             ; eax = BigVar1[0]
sub      eax, [di+0]             ; eax = BigVar1[0] - BigVar2[0]
mov      [bx+0], eax             ; BigTotal[0] = eax

; sottrazione delle seconde DWORD con SBB

mov      eax, [si+4]             ; eax = BigVar1[4]
sbb      eax, [di+4]             ; eax = BigVar1[4] - BigVar2[4] - CF
mov      [bx+4], eax             ; BigTotal[4] = eax

; sottrazione delle terze DWORD con SBB

mov      eax, [si+8]             ; eax = BigVar1[8]
sbb      eax, [di+8]             ; eax = BigVar1[8] - BigVar2[8] - CF
mov      [bx+8], eax             ; BigTotal[8] = eax

call     showCPU                 ; verifica dei flags
```

Dopo l'esecuzione dell'ultima sottrazione (tra le due **DWORD** piu' significative), consultiamo **CF** per sapere se c'e' stato un prestito finale; nel nostro caso, **CF=0**, per cui il risultato e' valido cosi' com'e'.

Se vogliamo visualizzare il risultato con **writeHex32**, possiamo procedere esattamente come per l'esempio riferito all'istruzione **ADC**.

17.4.2 Sottrazione tra numeri interi con segno di ampiezza arbitraria

Naturalmente, l'esempio appena illustrato e' perfettamente valido anche quando il minuendo e il sottraendo codificano numeri interi con segno a **96** bit; in tal caso, dobbiamo tenere presente che i numeri stessi risultano rappresentati in complemento a **2**, modulo $\mathbf{2^{96}}$. In base al concetto di **estensione del bit di segno**, possiamo dire che, tutti i numeri esadecimali a **96** bit, la cui cifra piu' significativa e' compresa tra **0** e **7** (cioe', tra **0000b** e **0111b**), sono positivi; tutti i numeri esadecimali a **96** bit, la cui cifra piu' significativa e' compresa tra **8** e **F** (cioe', tra **1000b** e **1111b**), sono negativi.

Per sottrarre tra loro numeri interi con segno di ampiezza arbitraria, si procede esattamente come mostrato nel precedente esempio; la prima sottrazione viene eseguita con **SUB**, mentre le sottrazioni successive vengono eseguite con **SBB**. Dopo l'esecuzione dell'ultima sottrazione (tra le due **DWORD** piu' significative), si deve consultare, non **CF**, ma bensi' **OF** e **SF**; e' chiaro, infatti, che il segno del risultato si trova codificato nel suo bit piu' significativo.
Se, dopo l'ultima sottrazione, si ha **OF=0**, allora il risultato e' valido, e il relativo segno si trova codificato in **SF**; se, invece, dopo l'ultima sottrazione, si ha **OF=1**, il risultato e' andato in overflow (in tal caso, il contenuto di **SF** non ha, ovviamente, nessun significato).

17.4.3 Effetti provocati da SBB sugli operandi e sui flags

L'esecuzione dell'istruzione **SBB**, modifica il contenuto del solo operando **DEST**, che viene sovrascritto dal risultato della somma appena effettuata; il contenuto dell'operando **SRC** rimane inalterato.
Anche per **SBB**, bisogna prestare attenzione ai soliti casi del tipo:

```
sbb si, [si+03F8h]
```

Dopo l'esecuzione di questa istruzione, il contenuto del registro puntatore **SI** viene modificato.

L'esecuzione dell'istruzione **SBB**, modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** del **Flags Register**; il contenuto di questi campi, ha l'identico significato gia' illustrato per l'istruzione **SUB**.

17.5 Le istruzioni INC e DEC

Con il mnemonico **INC** si indica l'istruzione **INCrement by one** (incremento di uno); questa istruzione richiede un unico operando (**DEST**), il cui contenuto viene incrementato di **1**. Si ottiene quindi:

```
DEST = DEST + 1
```

Si puo' dire quindi che l'istruzione **INC** e' una sorta di **ADD** che viene usata quando l'operando sorgente vale **1**; in un caso del genere, l'utilizzo di **INC** puo' portare alla generazione di un codice macchina piu' compatto ed efficiente rispetto a quello prodotto da **ADD**.
Non essendo possibile, ovviamente, incrementare un valore immediato, le uniche forme lecite per l'istruzione **INC** sono le seguenti:

```
INC      Reg
INC      Mem
```

Il codice macchina generale per l'istruzione **INC** e' rappresentato dall'**Opcode 1111111w**, seguito dal campo **mod_000_r/m**; se l'operando e' un registro, viene utilizzato un codice macchina formato dal solo **Opcode 01000_reg**.
E' chiaro quindi che, possibilmente, conviene utilizzare **INC** con un operando registro; in questo modo infatti, viene generato un codice macchina da **1** byte, che verra' eseguito piu' velocemente

dalla **CPU**.

Con il mnemonico **DEC** si indica l'istruzione **DECrement by one** (decremento di uno); questa istruzione richiede un unico operando (**DEST**), il cui contenuto viene decrementato di **1**. Si ottiene quindi:

```
DEST = DEST - 1
```

Si puo' dire quindi che l'istruzione **DEC** e' una sorta di **SUB** che viene usata quando l'operando sorgente (sottraendo) vale **1**; in un caso del genere, l'utilizzo di **DEC** puo' portare alla generazione di un codice macchina piu' compatto ed efficiente rispetto a quello prodotto da **SUB**.
Non essendo possibile, ovviamente, decrementare un valore immediato, le uniche forme lecite per l'istruzione **DEC** sono le seguenti:

```
DEC      Reg
DEC      Mem
```

Il codice macchina generale per l'istruzione **DEC** e' rappresentato dall'**Opcode 1111111w**, seguito dal campo **mod_001_r/m**; se l'operando e' un registro, viene utilizzato un codice macchina formato dal solo **Opcode 01001_reg**.
E' chiaro quindi che, possibilmente, conviene utilizzare **DEC** con un operando registro; in questo modo infatti, viene generato un codice macchina da **1** byte, che verra' eseguito piu' velocemente dalla **CPU**.

17.5.1 Effetti provocati da INC e DEC sugli operandi e sui flags

L'esecuzione delle istruzioni **INC** e **DEC**, modifica il contenuto dell'unico operando **DEST**, che viene sovrascritto dal risultato della operazione appena effettuata.

L'esecuzione dell'istruzione **INC** provoca la modifica dei flags **PF**, **AF**, **ZF**, **SF**, **OF**; come si puo' notare, a differenza di quanto accade con **ADD**, il flag **CF** non viene modificato!
Questo significa che se, ad esempio, **AX** contiene il valore **FFFFh**, e **CF=0**, allora l'istruzione:

```
inc ax
```

produce il seguente effetto:

```
AX = FFFFh + 0001h = 0000h, con CF = 0
```

Dopo l'esecuzione di questa istruzione, il registro **AX** conterra' quindi il valore **0000h**, mentre **CF** restera' inalterato in quanto il riporto provocato da **INC** viene ignorato.
Il fatto che **INC** non modifichi **CF**, e' frutto di una precisa scelta fatta dai progettisti delle **CPU**; infatti, in questo modo e' possibile utilizzare **INC** per incrementare il contatore di un **loop** (iterazione) senza interferire sul contenuto di **CF**.
Puo' capitare, pero', che il programmatore abbia la necessita' di provocare la modifica di **CF** in seguito ad una addizione il cui secondo addendo vale **1**; in un caso del genere, si deve per forza usare **ADD** scrivendo ad esempio:

```
add ax, 1
```

Per quanto riguarda il significato degli altri flags modificati da **INC**, la situazione e' del tutto identica al caso dell'istruzione **ADD** con operando **SRC** che vale **1**.

L'esecuzione dell'istruzione **DEC** provoca la modifica dei flags **PF**, **AF**, **ZF**, **SF**, **OF**; come si puo' notare, a differenza di quanto accade con **SUB**, il flag **CF** non viene modificato!
Questo significa che se, ad esempio, **AX** contiene il valore **0000h**, e **CF=0**, allora l'istruzione:

```
dec ax
```

produce il seguente effetto:

```
AX = 0000h - 0001h = FFFFh, con CF = 0
```

Dopo l'esecuzione di questa istruzione, il registro **AX** conterra' quindi il valore **FFFFh**, mentre **CF**

restera' inalterato in quanto il prestito richiesto da **DEC** viene ignorato.
Il fatto che **DEC** non modifichi **CF**, e' frutto di una precisa scelta fatta dai progettisti delle **CPU**; infatti, in questo modo e' possibile utilizzare **DEC** per decrementare il contatore di un **loop** (iterazione) senza interferire sul contenuto di **CF**.
Puo' capitare, pero', che il programmatore abbia la necessita' di provocare la modifica di **CF** in seguito ad una sottrazione il cui sottraendo vale **1**; in un caso del genere, si deve per forza usare **SUB** scrivendo ad esempio:

```
sub ax, 1
```

Per quanto riguarda il significato degli altri flags modificati da **DEC**, la situazione e' del tutto identica al caso dell'istruzione **SUB** con operando **SRC** che vale **1**.

17.6 L'istruzione NEG

Con il mnemonico **NEG** si indica l'istruzione **two's complement NEGation** (negazione di un numero intero con segno, espresso in complemento a **2**); questa istruzione richiede un unico operando (**DEST**), il cui contenuto viene trattato come numero intero con segno. Dopo l'esecuzione dell'istruzione **NEG**, l'operando **DEST** contiene il valore iniziale, cambiato pero' di segno; ad esempio, negando il numero negativo **-2500**, si ottiene il suo opposto, e cioe', il numero positivo **+2500**.
Non essendo possibile, ovviamente, negare un valore immediato, le uniche forme lecite per l'istruzione **NEG** sono le seguenti:

```
NEG      Reg
NEG      Mem
```

17.6.1 Complemento a 1 e complemento a 2

Nei capitoli dedicati alla matematica del computer, abbiamo visto come si deve procedere per negare un numero intero con segno; supponiamo di lavorare sugli interi con segno a **16** bit (compresi quindi tra i limiti **-32768** e **+32767**) e vediamo come si ottiene, ad esempio, la negazione del numero **+30541**. In matematica, il metodo piu' ovvio che si puo' applicare per negare un numero **n**, consiste nell'eseguire la sottrazione:

```
0 - n
```

Nel nostro caso, possiamo scrivere quindi:

```
0 - 30541 = -30541
```

Nell'aritmetica modulare del computer, per gli interi con segno a **16** bit in complemento a **2**, il valore **0** equivale a **65536** (**2^{16}**); possiamo scrivere quindi:

$0 - 30541 = 2^{16} - 30541 = 34995 = 88B3h$

Possiamo dire quindi che **34995** (cioe', **88B3h**) e' la codifica a **16** bit in complemento a **2**, del numero negativo **-30541**.
Per poter eseguire la precedente sottrazione, possiamo procedere in questo modo:

$2^{16} - 30541 = 65536 - 30541 = (65535 + 1) - 30541 = (65535 - 30541) + 1$

Traducendo tutto in binario, e svolgendo le varie operazioni otteniamo:

```
(65535 - 30541) + 1 =

1111111111111111b -
0111011101001101b =
-------------------
1000100010110010b +
0000000000000001b =
-------------------
1000100010110011b    (= 88B3h)
```

Osserviamo subito che la prima sottrazione, non fa' altro che invertire tutti i bit del numero da negare; infatti:
```
1111111111111111b - 0111011101001101b = 1000100010110010b
```
Come sappiamo, questa fase prende il nome di **complemento a 1**; per effettuare il complemento a **1** di un numero **n** (cioe', l'inversione di tutti i bit di **n**), la **CPU** mette a disposizione l'istruzione **NOT**, che verra' analizzata in un prossimo capitolo.
La seconda fase, che consiste nel sommare **1** al risultato della sottrazione, prende il nome di **complemento a 2**; possiamo dire allora che:
```
NEG(n) = NOT(n) + 1
```
Questo e' proprio il procedimento che la **CPU** utilizza per negare un numero intero con segno.

Applicando ora questo stesso metodo per negare **-30541** (cioe', **1000100010110011b**), dovremmo ottenere:
```
-(-30541) = +30541
```

Infatti:

```
1111111111111111b -
1000100010110011b =
-------------------
0111011101001100b +
0000000000000001b =
-------------------
0111011101001101b    (= 77D4h)
```

Traducendo **77D4h** in base **10**, otteniamo proprio **+30541**!

17.6.2 Negazione di numeri interi con segno di ampiezza arbitraria

Per negare un numero intero con segno di ampiezza arbitraria, possiamo utilizzare il metodo appena illustrato; a tale proposito, supponiamo di voler negare un numero formato da **96** bit (**3 DWORD**).
Prima di tutto, invertiamo tutti bit delle **3 DWORD**; successivamente, sommiamo **1** al risultato appena ottenuto.
Per effettuare la somma, ci serviamo dello stesso procedimento illustrato per l'istruzione **ADC**; quando avremo a disposizione l'istruzione **NOT**, vedremo un esempio pratico.

17.6.3 Effetti provocati da NEG sugli operandi e sui flags

L'esecuzione dell'istruzione **NEG**, modifica il contenuto dell'unico operando **DEST**, che viene sovrascritto dal risultato della negazione.

L'esecuzione dell'istruzione **NEG** provoca la modifica dei flags **CF**, **PF**, **AF**, **ZF**, **SF**, **OF**; per capire il significato di questi flags, bisogna ricordare che la negazione di un numero **n**, equivale allo svolgimento della sottrazione:
```
0 - n
```
Si deduce allora che se **n=0**, otteniamo:
```
0 - 0 = 0
```
Questa sottrazione non richiede, ovviamente, nessun prestito, per cui **CF=0**
Se, invece, il numero **n** da negare e' diverso da zero, si ottiene sempre **CF=1**; infatti, per negare, ad esempio, **+15800**, dobbiamo scrivere:
```
0 - 15800 = -15800
```
Questa sottrazione richiede, ovviamente, un prestito.

Anche per quanto riguarda gli altri flags, la situazione e' abbastanza chiara; e' sufficiente applicare tutte le considerazioni gia' svolte per l'istruzione **SUB** (con minuendo uguale a **0**).
Analizziamo, in particolare, il significato del flag **OF**; a tale proposito, supponiamo di voler operare sugli interi con segno a **16** bit, in complemento a **2**, compresi quindi tra **-32768** e **+32767**.
Negando un numero strettamente positivo, compreso quindi tra **+1** e **+32767**, otteniamo, come previsto, un numero negativo compreso tra **-1** e **-32767**; la **CPU** pone quindi **OF=0**. Viceversa, negando un numero negativo, compreso quindi tra **-1** e **-32767**, otteniamo, come previsto, un numero positivo compreso tra **+1** e **+32767**; la **CPU** pone quindi **OF=0**
Il numero negativo piu' piccolo, e cioe' **-32768**, rappresenta un caso particolare; infatti:
`0 - (-32768) = 0 + 32768 = 32768`
Ma per gli interi con segno a **16** bit in complemento a **2**, **32768** e' la codifica dello stesso numero negativo **-32768**; in sostanza, abbiamo ottenuto un numero "troppo" positivo che sconfina nell'insieme dei numeri negativi, per cui la **CPU** pone **OF=1**.

17.7 L'istruzione CMP

Con il mnemonico **CMP** si indica l'istruzione **CoMPare two operands** (comparazione tra due operandi); questa istruzione ha lo scopo di confrontare il contenuto dell'operando **SRC** con il contenuto dell'operando **DEST**, in modo da stabilire la relazione d'ordine che esiste tra i due operandi. In sostanza, grazie a **CMP** possiamo sapere se **DEST** e' maggiore, minore o uguale a **SRC**.
Il confronto viene effettuato attraverso la sottrazione:
`Temp = DEST - SRC`
In base al risultato della sottrazione, appare evidente che:

***** se **Temp** e' uguale a zero, allora **DEST** e' uguale a **SRC**;
***** se **Temp** e' maggiore di zero, allora **DEST** e' maggiore di **SRC**;
***** se **Temp** e' minore zero, allora **DEST** e' minore di **SRC**.

Subito dopo aver eseguito la sottrazione, la **CPU** modifica gli stessi flags coinvolti dall'istruzione **SUB** (con gli stessi operandi **DEST** e **SRC**); come viene spiegato piu' avanti, attraverso la consultazione di questi flags possiamo conoscere il risultato della comparazione.
Osserviamo che il risultato della sottrazione, viene disposto in un registro temporaneo della **CPU**, e questo significa che, a differenza di quanto accade con **SUB**, l'istruzione **CMP** preserva il contenuto, non solo di **SRC**, ma anche di **DEST**; e' chiaro, infatti, che lo scopo di **CMP** e' solo quello di confrontare due operandi, senza provocare su di essi nessuna modifica.

Le uniche forme lecite per l'istruzione **CMP** sono le seguenti:

```
CMP      Reg, Reg
CMP      Reg, Mem
CMP      Mem, Reg
CMP      Reg, Imm
CMP      Mem, Imm
```

Gli operandi **SRC** e **DEST** devono essere specificati esplicitamente dal programmatore, e devono avere la stessa ampiezza in bit; un eventuale operando **SRC** di tipo **Imm**, viene esteso attraverso il suo bit di segno, sino a raggiungere l'ampiezza in bit dell'operando **DEST**.

L'istruzione **CMP**, essendo formalmente identica a **SUB**, opera, indifferentemente, sui numeri interi con o senza segno; analizziamo in dettaglio il procedimento che bisogna seguire per determinare, nei due casi, il risultato della comparazione.

17.7.1 Comparazione tra numeri interi senza segno

Come e' stato detto in precedenza, l'esecuzione dell'istruzione **CMP** provoca la modifica degli stessi flags coinvolti da **SUB**, e cioe', **CF**, **PF**, **AF**, **ZF**, **SF**, **OF**; il contenuto di questi flags ci fornisce il risultato della comparazione.

Poniamo **AX=45200**, **BX=38260**, ed eseguiamo l'istruzione:

```
cmp ax, bx
```

In binario si ha:

```
AX = 45200 = 1011000010010000b
```

e:

```
BX = 38260 = 1001010101110100b
```

La **CPU** esegue quindi la sottrazione:

```
1011000010010000b -
1001010101110100b =
-------------------
0001101100011100b     (6940)
```

Sulla base di questo risultato, la **CPU** pone:

```
CF = 0, PF = 0, AF = 1, ZF = 0, SF = 0, OF = 0
```

Per giustificare questa situazione, osserviamo innanzi tutto che il risultato e' diverso da **0**, per cui **ZF=0**; cio' significa che, indipendentemente dal contenuto degli altri flags, **DEST** e' sicuramente **diverso** da **SRC** (altrimenti avremmo ottenuto **ZF=1**).
Nell'insieme degli interi senza segno, **1011000010010000b** rappresenta **45200**, mentre **1001010101110100b** rappresenta **38260**; la sottrazione:

```
45200 - 38260 = 6940
```

produce un risultato positivo (minuendo maggiore o uguale al sottraendo), per cui **CF=0** (nessun prestito).

Cio' significa che, indipendentemente dal contenuto degli altri flags, **DEST** e' sicuramente **non minore** di **SRC**; se il minuendo fosse stato minore del sottraendo, avremmo ottenuto **CF=1** (prestito), e quindi, indipendentemente dal contenuto degli altri flags, **DEST** sarebbe stato sicuramente **non maggiore** di **SRC**.

A questo punto, e' perfettamente inutile mostrare altri esempi, in quanto abbiamo gia' capito che i due soli flags **ZF** e **CF**, ci forniscono in modo inequivocabile, il risultato di una qualsiasi comparazione tra numeri interi senza segno; per rappresentare i vari risultati possibili, ci serviamo dei seguenti simboli (presi in prestito dal linguaggio **C**):

Simbolo	Significato
<	strettamente minore
>	strettamente maggiore
==	uguale
!=	diverso
<=	minore o uguale
>=	maggiore o uguale

Utilizzando questi simboli, possiamo affermare che:

ZF	CF	Risultato della comparazione
1	x	DEST == SRC (indipendentemente dal contenuto di CF)
1	x	DEST >= SRC (indipendentemente dal contenuto di CF)
1	x	DEST <= SRC (indipendentemente dal contenuto di CF)
0	x	DEST != SRC (indipendentemente dal contenuto di CF)
0	0	DEST > SRC
0	1	DEST < SRC

Dalle considerazioni appena esposte, si puo' notare che il metodo per ricavare il risultato della comparazione tra numeri interi senza segno, e' piuttosto semplice; in ogni caso, e' importante capire bene il significato di concetti come, **minore o uguale**, **maggiore o uguale**, **non minore**, **non maggiore**, etc.
Se **ZF=1**, allora **DEST** e' sicuramente **UGUALE** a **SRC**; possiamo anche dire che **DEST** e' **NON DIVERSO** da **SRC**, oppure che **DEST** e' **MAGGIORE O UGUALE** a **SRC**, oppure che **DEST** e' **MINORE O UGUALE** a **SRC**, etc.
Analogamente, se **ZF=0**, allora **DEST** e' sicuramente **DIVERSO** da **SRC**; possiamo anche dire che **DEST** e' **NON UGUALE** a **SRC**.
Se **ZF=0** e **CF=0**, allora **DEST** e' sicuramente **MAGGIORE** di **SRC**; possiamo anche dire che **DEST** e' **NON MINORE NE' UGUALE** a **SRC**.
Se **ZF=0** e **CF=1**, allora **DEST** e' sicuramente **MINORE** di **SRC**; possiamo anche dire che **DEST** e' **NON MAGGIORE NE' UGUALE** a **SRC**.
Tutti questi aspetti verranno ulteriormente chiariti in un apposito capitolo.

17.7.2 Comparazione tra numeri interi con segno

Per quanto riguarda la comparazione tra numeri interi con segno, la situazione diventa piu' impegnativa; in ogni caso, basta seguire la logica per rendersi conto che non c'e' niente di complesso.
Facciamo riferimento ai numeri interi con segno a **16** bit in complemento a **2**, compresi quindi tra i limiti **-32768** e **+32767**; come si puo' facilmente intuire, questa volta i flags che ci interessano sono **OF**, **SF** e **ZF**.
Complessivamente, si possono presentare quattro casi distinti, che vengono analizzati nel seguito; appare ovvio il fatto che, anche per i numeri interi con segno, il flag **ZF** indica se i due operandi sono uguali (**ZF=1**) o diversi (**ZF=0**).

1) La sottrazione **DEST - SRC** produce un risultato compreso tra **0** e **+32767**; cio' accade quando **DEST** e' "leggermente" maggiore o uguale a **SRC**. Vediamo alcuni esempi:

```
(+23500) - (+12800) = +10700                    OF=0, SF=0, ZF=0
(+32700) - (-67) = (+32700) + (+67) = +32767    OF=0, SF=0, ZF=0
(+12350) - (+12350) = 0                         OF=0, SF=0, ZF=1
```

Come si puo' notare, in questo caso, non essendo possibile l'overflow, il flag **OF** vale sempre **0**; anche **SF** vale sempre **0** in quanto **DEST** e' maggiore o uguale a **SRC**, e quindi la sottrazione produce sempre un risultato non negativo. In definitiva, possiamo dire che in questo caso, **OF** e **SF** sono sempre uguali tra loro e valgono entrambi **0**.

2) La sottrazione **DEST - SRC** produce un risultato compreso tra **+32768** e **+65535**; cio' accade quando **DEST** e' "eccessivamente" maggiore di **SRC**. Vediamo alcuni esempi:

```
(+28500) - (-15000) = (+28500) + (+15000) = +43500    OF=1, SF=1, ZF=0
0 - (-32768) = 0 + (+32768) = +32768                  OF=1, SF=1, ZF=0
(+32767) - (-32768) = (+32767) + (+32768) = +65535    OF=1, SF=1, ZF=0
```

In pratica, **DEST** e' strettamente maggiore di **SRC**, ma la loro differenza produce un numero "troppo" positivo che sconfina nell'insieme dei numeri negativi; il risultato va' quindi in overflow, per cui si avra' sempre **OF=1**. Anche **SF** vale sempre **1**, in quanto la sottrazione produce sempre un risultato che sconfina nell'insieme dei numeri negativi; in definitiva, possiamo dire che in questo caso, **OF** e **SF** sono sempre uguali tra loro e valgono entrambi **1**.

3) La sottrazione **DEST - SRC** produce un risultato compreso tra **-32768** e **-1**; cio' accade quando **DEST** e' "leggermente" minore di **SRC**. Vediamo alcuni esempi:

```
(-10) - (-9) = (-10) + (+9) = -1    OF=0, SF=1, ZF=0
(+18000) - (+19000) = -1000         OF=0, SF=1, ZF=0
(-32768) - 0 = -32768               OF=0, SF=1, ZF=0
```

Come si puo' notare, in questo caso, non essendo possibile l'overflow, il flag **OF** vale sempre **0**; il flag **SF**, invece, vale sempre **1** in quanto **DEST** e' minore di **SRC**, e quindi la sottrazione produce sempre un risultato negativo. In definitiva, possiamo dire che in questo caso, **OF** e **SF** sono sempre diversi tra loro, in quanto assumono, rispettivamente, i valori **0** e **1**.

4) La sottrazione **DEST - SRC** produce un risultato compreso tra **-65535** e **-32769**; cio' accade quando **DEST** e' "eccessivamente" minore di **SRC**. Vediamo alcuni esempi:

```
(-11000) - (+30000) = -41000    OF=1, SF=0, ZF=0
(-32768) - (+1) = -32769        OF=1, SF=0, ZF=0
(-32768) - (+32767) = -65535    OF=1, SF=0, ZF=0
```

In pratica, **DEST** e' strettamente minore di **SRC**, ma la loro differenza produce un numero "troppo" negativo, che sconfina nell'insieme dei numeri positivi; il risultato va' quindi in overflow, per cui si avra' sempre **OF=1**. Il flag **SF**, invece, vale sempre **0**, in quanto la sottrazione produce sempre un risultato che sconfina nell'insieme dei numeri positivi; in definitiva, possiamo dire che in questo caso, **OF** e **SF** sono sempre diversi tra loro, in quanto assumono, rispettivamente, i valori **1** e **0**.

Sulla base delle considerazioni appena esposte, appare chiaro il metodo che bisogna seguire per ricavare dai flags il risultato della comparazione tra due numeri interi con segno; in pratica, possiamo affermare che:

ZF	OF	SF	Risultato della comparazione
1	x	x	DEST == SRC (indipendentemente dal contenuto di OF e SF)
1	x	x	DEST >= SRC (indipendentemente dal contenuto di OF e SF)
1	x	x	DEST <= SRC (indipendentemente dal contenuto di OF e SF)
0	x	x	DEST != SRC (indipendentemente dal contenuto di OF e SF)
0	0	0	DEST > SRC
0	1	1	DEST > SRC
0	0	1	DEST < SRC
0	1	0	DEST < SRC

Questa tabella puo' essere ulteriormente semplificata osservando che, se **ZF=1**, allora **DEST** e' sicuramente uguale a **SRC**, e cio' vale indipendentemente dal contenuto di **OF** e **SF**; se, invece, **ZF=0**, allora **DEST** e' sicuramente diverso da **SRC**, e cio' vale indipendentemente dal contenuto di **OF** e **SF**. In particolare, se **ZF=0** e **OF** e' uguale a **SF**, allora **DEST** e' sicuramente maggiore di **SRC**; se, invece, **ZF=0** e **OF** e' diverso da **SF**, allora **DEST** e' sicuramente minore di **SRC**.

17.7.3 Istruzioni per il controllo del flusso

Le considerazioni appena esposte sono molto semplici da un punto di vista teorico, ma piuttosto scomode da applicare in pratica; infatti, sarebbe una follia pensare di svolgere tutti questi controlli ogni volta che dobbiamo effettuare una banale comparazione numerica.
Fortunatamente, pero', la **CPU** ci viene incontro mettendoci a disposizione una miriade di istruzioni che eseguono tutto questo lavoro al nostro posto; si tratta delle cosiddette **istruzioni per il controllo del flusso** (o **istruzioni per il trasferimento del controllo**) che, come e' stato spiegato nel capitolo dedicato alle reti combinatorie, hanno una importanza enorme nella programmazione.
Attraverso queste istruzioni, e' possibile scrivere programmi "intelligenti", capaci di prendere decisioni in base al risultato di una comparazione numerica (o di altri tipi di test); le istruzioni per il trasferimento del controllo, verranno illustrate in un apposito capitolo.
Utilizzando queste stesse istruzioni, i linguaggi di alto livello implementano particolari costrutti sintattici, come le istruzioni condizionali **IF**, **THEN**, **ELSE**, i cicli **FOR**, i cicli **WHILE DO**, i cicli **REPEAT UNTIL**, etc; considerando, ad esempio, due variabili numeriche **A** e **B**, possiamo scrivere la seguente sequenza di istruzioni in pseudo-linguaggio **C**:

```
if (A > B)                        /* se A e' maggiore di B */
   esegui il blocco istruzioni 1;
else if (A < B)                   /* altrimenti, se A e' minore di B */
   esegui il blocco istruzioni 2;
else                              /* altrimenti, se A e' uguale B */
   esegui il blocco istruzioni 3;
```

Come si puo' notare, questo programma e' in grado di compiere tre scelte distinte, in base al risultato della comparazione tra **A** e **B**; il codice macchina di questo programma, generato dal compilatore **C**, utilizza proprio l'istruzione **CMP** per effettuare le varie comparazioni.

17.7.4 Comparazione tra numeri interi di ampiezza arbitraria

Un'ultima considerazione sull'istruzione **CMP**, riguarda il caso in cui si abbia la necessita' di dover comparare tra loro numeri interi di ampiezza arbitraria; in tal caso, il metodo da seguire e' molto semplice, e consiste nel mettere assieme le cose dette per le due istruzioni **SBB** e **CMP**.
In pratica, prima di tutto si sottraggono i due operandi attraverso il metodo illustrato per l'istruzione **SBB**; si puo' notare che tale metodo, preserva il contenuto di entrambi gli operandi. Terminata la sottrazione, se si sta operando sui numeri interi senza segno, si consultano i flags **CF** e **ZF**; se, invece, si sta operando sui numeri interi con segno, si consultano i flags **OF**, **SF** e **ZF**.

17.7.5 Effetti provocati da CMP sugli operandi e sui flags

L'esecuzione dell'istruzione **CMP**, preserva il contenuto, sia dell'operando **SRC**, sia dell'operando **DEST**; infatti, il risultato della sottrazione tra **DEST** e **SRC** viene ignorato.

L'esecuzione dell'istruzione **CMP**, modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** del **Flags Register**; il contenuto di questi campi, ha l'identico significato gia' illustrato per l'istruzione **SUB**.

17.8 Le istruzioni CBW, CWD, CWDE, CDQ

Nel precedente capitolo abbiamo visto che con le **CPU 80386** e superiori, attraverso le due istruzioni **MOVSX** e **MOVZX** e' possibile effettuare un trasferimento dati da un operando sorgente avente una certa ampiezza in bit, ad un operando destinazione avente ampiezza maggiore; l'istruzione **MOVZX** opera sui numeri interi senza segno, mentre l'istruzione **MOVSX** opera sui numeri interi con segno.
Se si utilizza **MOVZX**, il valore da trasferire viene sempre trattato come numero intero senza segno (**unsigned**), e viene "allungato" a sinistra con degli zeri (**zero extension**); se si utilizza **MOVSX**, il valore da trasferire viene sempre trattato come numero intero con segno (**signed**) in complemento a **2**, e viene "allungato" a sinistra attraverso il suo bit di segno (**sign extension**).

Tutte le **CPU** della famiglia **80x86**, dispongono anche di una serie di istruzioni aritmetiche che permettono di estendere l'ampiezza in bit del valore contenuto in un operando; tale operando, viene sempre trattato come numero intero con segno in complemento a **2**, e viene quindi sottoposto all'estensione del suo bit di segno.

17.8.1 L'istruzione CBW

Con il mnemonico **CBW** si indica l'istruzione **Convert Byte to Word** (conversione di un **BYTE** in una **WORD**); l'unica forma lecita per questa istruzione, e':

```
CBW
```

Il valore da convertire deve trovarsi obbligatoriamente nel registro (implicito) **AL**; tale valore, viene trattato sempre come numero intero con segno a **8** bit in complemento a **2**.
Quando la **CPU** incontra l'istruzione **CBW**, legge il bit piu' significativo di **AL**, e lo copia in tutti gli **8** bit di **AH**; il risultato ottenuto dalla **CPU** si trova quindi nel registro **AX**.

Supponiamo, ad esempio, di avere **AL=01001110b** (**+78**); in presenza dell'istruzione:

```
cbw
```

la **CPU** legge il bit piu' significativo di **AL** (**0**), e lo copia in tutti gli **8** bit di **AH**. Si ottiene quindi:

```
AX = 0000000001001110b = 78
```

Come si puo' notare, abbiamo ottenuto la rappresentazione a **16** bit in complemento a **2**, del numero intero con segno **+78**.

Supponiamo, ad esempio, di avere **AL=11101110b** (**-18**); in presenza dell'istruzione:

```
cbw
```

la **CPU** legge il bit piu' significativo di **AL** (**1**), e lo copia in tutti gli **8** bit di **AH**. Si ottiene quindi:

```
AX = 1111111111101110b = 65518
```

Come si puo' notare, abbiamo ottenuto la rappresentazione a **16** bit in complemento a **2**, del numero intero con segno **-18**; infatti:

$65518 - 2^{16} = -18$

17.8.2 Le istruzioni CWD e CWDE

Con il mnemonico **CWD** si indica l'istruzione **Convert Word to Doubleword** (conversione di una **WORD** in una **DWORD**); l'unica forma lecita per questa istruzione, e':

```
CWD
```

Il valore da convertire deve trovarsi obbligatoriamente nel registro (implicito) **AX**; tale valore, viene trattato sempre come numero intero con segno a **16** bit in complemento a **2**.
Quando la **CPU** incontra l'istruzione **CWD**, legge il bit piu' significativo di **AX**, e lo copia in tutti i **16** bit di **DX**; il risultato ottenuto dalla **CPU**, si trova quindi nella coppia di registri **DX:AX**, con **DX** che, nel rispetto della convenzione **little endian**, contiene la **WORD** piu' significativa del risultato.

Supponiamo, ad esempio, di avere **AX=0111111111111111b** (**+32767**); in presenza dell'istruzione:

```
cwd
```

la **CPU** legge il bit piu' significativo di **AX** (**0**), e lo copia in tutti i **16** bit di **DX**. Si ottiene quindi:

```
DX:AX = 00000000000000000111111111111111b = 32767
```

Come si puo' notare, abbiamo ottenuto la rappresentazione a **32** bit in complemento a **2**, del numero intero con segno **+32767**.

Supponiamo, ad esempio, di avere **AX=1000000000000000b** (**-32768**); in presenza dell'istruzione:

```
cwd
```

la **CPU** legge il bit piu' significativo di **AX** (**1**), e lo copia in tutti i **16** bit di **DX**. Si ottiene quindi:

```
DX:AX = 11111111111111111000000000000000b = 4294934528
```

Come si puo' notare, abbiamo ottenuto la rappresentazione a **32** bit in complemento a **2**, del numero intero con segno **-32768**; infatti:

$4294934528 - 2^{32} = -32768$

L'istruzione **CWD**, e' stata concepita per poter lavorare nel **modo 16 bit** delle **CPU 8086**; se vogliamo lavorare, esplicitamente, nel **modo 32 bit** delle **CPU 80386** e superiori, possiamo servirci dell'istruzione **CWDE**.
Con il mnemonico **CWDE** si indica l'istruzione **Convert Word to Doubleword - Extended form** (conversione di una **WORD** in una **DWORD**); l'unica forma lecita per questa istruzione, e':

```
CWDE
```

Il valore da convertire deve trovarsi obbligatoriamente nel registro (implicito) **AX**; tale valore, viene trattato sempre come numero intero con segno a **16** bit in complemento a **2**.
Quando la **CPU** incontra l'istruzione **CWDE**, legge il bit piu' significativo di **AX**, e lo copia in tutti i **16** bit piu' significativi di **EAX**; il risultato ottenuto dalla **CPU**, si trova quindi nel registro **EAX**.
Ripetendo allora con **CWDE**, i due esempi appena illustrati per **CWD**, otteniamo, direttamente in **EAX**, l'intero risultato finale a **32** bit.

17.8.3 L'istruzione CDQ

Con il mnemonico **CDQ** si indica l'istruzione **Convert Doubleword to Quadword** (conversione di una **DWORD** in una **QWORD**); l'unica forma lecita per questa istruzione, e':

```
CDQ
```

Il valore da convertire deve trovarsi obbligatoriamente nel registro (implicito) **EAX**; tale valore, viene trattato sempre come numero intero con segno a **32** bit in complemento a **2**.
Quando la **CPU** incontra l'istruzione **CDQ**, legge il bit piu' significativo di **EAX**, e lo copia in tutti i **32** bit di **EDX**; il risultato ottenuto dalla **CPU**, si trova quindi nella coppia di registri **EDX:EAX**, con **EDX** che, nel rispetto della convenzione **little endian**, contiene la **DWORD** piu' significativa del risultato.

Supponiamo, ad esempio, di avere **EAX=01110111001100110000110111001010b** (**+1999834570**); in presenza dell'istruzione:

```
cdq
```

la **CPU** legge il bit piu' significativo di **EAX** (**0**), e lo copia in tutti i **32** bit di **EDX**. Si ottiene quindi:

```
EDX:EAX = 0000000000000000000000000000000001110111001100110000110111001010b = 
1999834570
```

Come si puo' notare, abbiamo ottenuto la rappresentazione a **64** bit in complemento a **2**, del numero intero con segno **+1999834570**.

Supponiamo, ad esempio, di avere **EAX=11000111111011111110100111110110b** (**-940578314**); in presenza dell'istruzione:

```
cdq
```

la **CPU** legge il bit piu' significativo di **EAX** (**1**), e lo copia in tutti i **32** bit di **EDX**. Si ottiene quindi:

```
EDX:EAX = 1111111111111111111111111111111111000111111011111110100111110110b = 
18446744072768973302
```

Come si puo' notare, abbiamo ottenuto la rappresentazione a **64** bit in complemento a **2**, del numero intero con segno **-940578314**; infatti:

$18446744072768973302 - 2^{64} = -940578314$

Nota importante.
Osserviamo che, se vogliamo estendere a **32** bit l'ampiezza di un numero intero con segno a **8** bit in complemento a **2**, non dobbiamo fare altro che caricare il numero stesso in **AL**, ed eseguire poi, in sequenza, le due istruzioni **CBW** e **CWD** (oppure, **CBW** e **CWDE**); analogamente, se vogliamo estendere a **64** bit l'ampiezza di un numero intero con segno a **8** bit in complemento a **2**, non dobbiamo fare altro che caricare il numero stesso in **AL**, ed eseguire poi, in sequenza, le tre istruzioni **CBW**, **CWDE** e **CDQ**.
Infine, se vogliamo estendere a **64** bit l'ampiezza di un numero intero con segno a **16** bit in complemento a **2**, non dobbiamo fare altro che caricare il numero stesso in **AX**, ed eseguire poi, in sequenza, le due istruzioni **CWDE** e **CDQ**.

17.8.4 Estensione di numeri interi con segno di ampiezza arbitraria

Se vogliamo operare su numeri interi con segno di ampiezza arbitraria, non dobbiamo fare altro che applicare i concetti appena esposti; supponiamo, ad esempio, di voler estendere a **128** bit il seguente numero intero con segno a **64** bit:

```
BigNum64 dq 93B148F6C839ABF3h
```

Prima di tutto, creiamo una locazione di memoria da **128** bit con la seguente definizione:

```
BigNum128 dd 4 dup (0)
```

Nei **64** bit meno significativi di **BigNum128**, dobbiamo disporre, ovviamente, il contenuto di **BigNum64**; in tutti i **64** bit piu' significativi di **BigNum128**, dobbiamo copiare il bit di segno di **BigNum64**.
A tale proposito, possiamo applicare un trucco molto semplice, che consiste nell'utilizzare **CDQ** per estendere solo la **DWORD** piu' significativa (cioe', quella contenente il bit di segno) di **BigNum64**; come sappiamo, il risultato ottenuto da **CDQ** viene disposto nella coppia di registri **EDX:EAX**. Il registro **EDX** contiene quindi l'estensione del bit di segno di **BigNum64**; infatti, se **BigNum64** e' positivo (cifra piu' significativa compresa tra **0h** e **7h**), si ottiene **EDX=00000000h**, mentre se **BigNum64** e' negativo (cifra piu' significativa compresa tra **8h** e **Fh**), si ottiene **EDX=FFFFFFFFh**.

Per svolgere tutto questo lavoro, possiamo utilizzare il seguente codice:

```
; inizializzazione puntatori a BigNum64 e a BigNum128

mov      si, offset BigNum64        ; ds:si punta a BinNum64
mov      di, offset BigNum128       ; ds:di punta a BigNum128

; copia di BigNum64 nei primi 64 bit di BigNum128

mov      eax, [si+0]                ; eax = BigNum64[0]
mov      [di+0], eax                ; BigNum128[0] = eax
mov      eax, [si+4]                ; eax = BigNum64[4]
mov      [di+4], eax                ; BigNum128[4] = eax

; individuazione del bit di segno di BigNum64

cdq                                 ; (eax = BigNum64[4])

; estensione del bit di segno (EDX) di BigNum64

mov      [di+8], edx                ; BigNum128[8] = edx
mov      [di+12], edx               ; BigNum128[12] = edx
```

Osserviamo che, in relazione all'esecuzione di **CDQ**, il registro **EAX** contiene gia' il valore che ci interessa, e cioe', **BigNum64[4]**.
Se ora vogliamo visualizzare il risultato con **writeHex32**, possiamo utilizzare lo stesso metodo illustrato per l'istruzione **ADC**.

17.8.5 Effetti provocati da CBW, CWD, CWDE e CDQ, sugli operandi e sui flags

In base alle considerazioni appena svolte, risulta che l'esecuzione dell'istruzione **CBW**, modifica il contenuto del solo registro **AX**; l'esecuzione dell'istruzione **CWD**, modifica il contenuto dei registri **AX** e **DX**. L'esecuzione dell'istruzione **CWDE**, modifica il contenuto del solo registro **EAX**; l'esecuzione dell'istruzione **CDQ**, modifica il contenuto dei registri **EAX** e **EDX**.

Le istruzioni **CBW**, **CWD**, **CWDE** e **CDQ**, hanno il solo scopo di effettuare l'estensione del bit di segno del loro operando; di conseguenza, l'esecuzione di queste istruzioni non modifica nessun campo del **Flags Register**.

17.9 L'istruzione MUL

Con il mnemonico **MUL** si indica l'istruzione **unsigned MULtiplication of AL/AX/EAX** (moltiplicazione di un numero intero senza segno per **AL/AX/EAX**); le uniche forme lecite per l'istruzione **MUL**, sono le seguenti:

```
MUL      Reg
MUL      Mem
```

Come si puo' notare, il programmatore deve specificare, esplicitamente, uno solo dei due fattori (**SRC**); l'altro fattore e', implicitamente, **AL**, **AX** o **EAX**. A seconda dell'ampiezza in bit (**8**, **16** o **32**) dell'operando **SRC**, la **CPU** decide se moltiplicare, rispettivamente, per **AL**, per **AX** o per **EAX**; e' proibito l'uso di un operando **SRC** di tipo **Imm**.

Come abbiamo visto nel capitolo dedicato alla matematica del computer, moltiplicando tra loro due numeri interi senza segno da **n** bit ciascuno, otteniamo un risultato che occupa, al massimo, **2n** bit; inoltre, il prodotto massimo che si puo' ottenere con due fattori da **n** bit, e' nettamente inferiore al valore massimo rappresentabile con **2n** bit.
Considerando, ad esempio, numeri interi senza segno a **8** bit, il prodotto massimo che possiamo ottenere e':

```
255 * 255 = 65025
```

Il valore **65025** e' nettamente inferiore al valore massimo (**65535**) rappresentabile con **16** bit.

Le considerazioni appena svolte dimostrano, inoltre, che la moltiplicazione provoca un cambiamento di modulo; proprio per questo motivo, la **CPU** dispone di una istruzione che moltiplica numeri interi senza segno (**MUL**), e di una istruzione che moltiplica numeri interi con segno (**IMUL**).

Se vogliamo conoscere le convenzioni seguite dalla **CPU** per la memorizzazione del risultato della moltiplicazione eseguita con **MUL**, dobbiamo analizzare i tre casi fondamentali che si possono presentare.

17.9.1 Moltiplicazione tra numeri interi senza segno a 8 bit

Poniamo, **AL=190**, **BH=212**, ed eseguiamo l'istruzione:

```
mul bh
```

In presenza di questa istruzione, la **CPU** utilizza implicitamente **AL**, e calcola:

```
AX = AL * BH = 190 * 212 = 40280
```

Come si puo' notare, il risultato a **16** bit **40280**, viene disposto nel registro **AX**.

17.9.2 Moltiplicazione tra numeri interi senza segno a 16 bit

Poniamo, **AX=36850**, e supponiamo che **ES:DI** punti in memoria ad una **WORD** che vale **60321**; eseguiamo ora l'istruzione:

```
mul word ptr es:[di]
```

L'operatore **WORD PTR** e' necessario per indicare all'assembler che gli operandi sono a **16** bit; il segment override e' necessario perche' **ES** non e' il registro di segmento naturale per i dati.
In presenza di questa istruzione, la **CPU** utilizza implicitamente **AX**, e calcola:

```
DX:AX = AX * ES:[DI] = 36850 * 60321 = 2222828850
```

Come si puo' notare, il risultato a **32** bit **2222828850**, viene disposto nella coppia di registri **DX:AX**; nel rispetto della convenzione **little endian**, il registro **DX** contiene la **WORD** piu' significativa del risultato.

17.9.3 Moltiplicazione tra numeri interi senza segno a 32 bit

Poniamo, **EAX=1967850342**, **ECX=3997840212**, ed eseguiamo la seguente istruzione:
```
mul ecx
```
In presenza di questa istruzione, la **CPU** utilizza implicitamente **EAX**, e calcola:
```
EDX:EAX = EAX * ECX = 1967850342 * 3997840212 = 7867151228445552504
```
Come si puo' notare, il risultato a **64** bit **7867151228445552504**, viene disposto nella coppia di registri **EDX:EAX**; nel rispetto della convenzione **little endian**, il registro **EDX** contiene la **DWORD** piu' significativa del risultato.

17.9.4 Moltiplicazione tra numeri interi senza segno di ampiezza arbitraria

Per moltiplicare numeri interi senza segno di ampiezza arbitraria, possiamo servirci di un algoritmo del tutto simile a quello che si utilizza con carta e penna; per analizzare tale algoritmo, supponiamo di voler calcolare:
```
3CFEh * Ah = 261ECh
```

Osserviamo subito che il primo prodotto parziale e':
```
Ah * Eh = 8Ch
```
cioe' **Ch** con riporto pari a **8h**; il riporto **8h** verra' sommato al prodotto parziale successivo.

Il secondo prodotto parziale e':
```
Ah * Fh = 96h
```
cioe' **6h** con riporto pari a **9h**; alla cifra **6h** dobbiamo sommare il precedente riporto **8h**, ottenendo:
```
6h + 8h = Eh
```
Osserviamo che questa somma, potrebbe anche provocare un **carry** (**CF=1**); in tal caso, sarebbe necessario incrementare di **1** il riporto da sommare al prodotto parziale successivo.

Il terzo prodotto parziale e':
```
Ah * Ch = 78h
```
cioe' **8h** con riporto pari a **7h**; alla cifra **8h** dobbiamo sommare il precedente riporto **9h**, ottenendo:
```
8h + 9h = 11h
```
Questa volta, la somma ha provocato un **carry** (**CF=1**); il riporto **7h** da sommare al prodotto parziale successivo deve essere quindi incrementato di **1**, e diventa cosi' **8h**.

Il quarto prodotto parziale e':
```
Ah * 3h = 1Eh
```
cioe' **Eh** con riporto pari a **1h**; alla cifra **Eh** dobbiamo sommare il precedente riporto **8h**, ottenendo:
```
Eh + 8h = 16h
```
Anche questa volta, la somma ha provocato un **carry** (**CF=1**); il riporto **1h** da sommare al prodotto parziale successivo deve essere quindi incrementato di **1**, e diventa cosi' **2h**.

Non esistendo altri prodotti parziali da eseguire, l'ultimo riporto **2h** rappresenta la cifra piu' significativa del risultato; unendo le cinque cifre che abbiamo ricavato nei calcoli precedenti, otteniamo proprio il prodotto finale **261ECh**.

In base alle considerazioni appena esposte, resta un solo dubbio legato al caso in cui la somma tra un prodotto parziale e il riporto precedente, provochi un **carry**; se si verifica questa eventualita', abbiamo visto che dobbiamo incrementare di **1** la cifra piu' significativa dello stesso prodotto parziale. La domanda che allora ci poniamo e': questo incremento di **1** puo' provocare un ulteriore **carry**?
La risposta e' no!

Infatti, il prodotto parziale piu' grande che possiamo ottenere e':

```
Fh * Fh = E1h
```

Se il riporto precedente era **Fh** (cioe', il piu' grande possibile), dobbiamo sommarlo alla cifra meno significativa del prodotto parziale **E1h**, ottenendo:

```
Fh + 1h = 10h
```

Come si puo' notare, questa somma ha provocato un **carry**, per cui dobbiamo incrementare di **1** la cifra piu' significativa del prodotto parziale **E1h**, ottenendo:

```
Eh + 1h = Fh
```

Possiamo dire quindi che l'eventualita' che si verifichi un ulteriore **carry** e' assolutamente impossibile!

Una volta chiariti questi dettagli, possiamo passare alla implementazione pratica dell'algoritmo per la moltiplicazione tra numeri interi senza segno di ampiezza arbitraria; supponendo di avere a disposizione una **CPU 80386** o superiore, ci serviremo dell'istruzione **MUL** con operandi di tipo **DWORD**.
In tal caso, ogni **DWORD** equivale ad una delle cifre esadecimali dell'esempio che abbiamo svolto in precedenza; in particolare, osserviamo che la **CPU**, nell'eseguire l'istruzione **MUL**, si serve, implicitamente, del registro **EAX**, e ci restituisce i vari prodotti parziali nella coppia **EDX:EAX**. La **DWORD** contenuta in **EDX** rappresenta il riporto destinato al prodotto parziale successivo, mentre la **DWORD** contenuta in **EAX**, deve essere sommata con l'eventuale riporto precedente; il risultato di questa somma rappresenta una delle **DWORD** del prodotto finale, e deve essere quindi salvato in una apposita locazione di memoria.
Come sappiamo, la somma tra **EAX** e l'eventuale riporto precedente, puo' provocare un **carry**; in tal caso, dobbiamo incrementare di **1** la **DWORD** contenuta in **EDX**.
Per effettuare questo incremento, possiamo servirci dell'istruzione:

```
adc edx, 0
```

Come si puo' facilmente constatare, questa istruzione calcola:

```
EDX = EDX + 0 + CF
```

Di conseguenza, se **CF=0**, allora **EDX** rimane invariato; se, invece, **CF=1**, allora **EDX** viene incrementato di **1**.
In base a quanto e' stato dimostrato in precedenza, questo incremento di **EDX** non puo' mai provocare un ulteriore **carry**; infatti, il prodotto massimo tra operandi di tipo **DWORD** e':

```
FFFFFFFFh * FFFFFFFFh = FFFFFFFE00000001h
```

Otteniamo quindi **00000001h** con riporto pari a **FFFFFFFEh**; se il riporto precedente era **FFFFFFFFh** (cioe', il piu' grande possibile), dobbiamo eseguire la somma:

```
00000001h + FFFFFFFFh = 100000000h
```

Questa somma provoca un **carry**, per cui dobbiamo incrementare di **1** il riporto **FFFFFFFEh**, ottenendo:

```
FFFFFFFEh + 1h = FFFFFFFFh
```

Un ulteriore **carry** e' quindi assolutamente impossibile.

Anche il contenuto di **EDX** deve essere salvato da qualche parte; infatti, la coppia **EDX:EAX** viene sovrascritta dal prodotto parziale successivo.
In analogia con l'esempio svolto in precedenza, possiamo dire che l'ultimo riporto che otteniamo, rappresenta la **DWORD** piu' significativa del prodotto finale.

A questo punto possiamo procedere con un esempio pratico; a tale proposito, supponiamo di voler calcolare:
3AC926F7914388C21F8694EDh * 8BA9F6C5h = 20123F9DAD8B72F05DA7AA6295215861h

Moltiplicando un numero a **96** bit per un numero a **32** bit, si ottiene un risultato a **96+32=128** bit; nel blocco dati del nostro programma, possiamo inserire allora le seguenti definizioni:

```
moltiplicando   dd    1F8694EDh, 914388C2h, 3AC926F7h
moltiplicatore dd    8BA9F6C5h
prodotto       dd    4 dup (0)
```

Il codice che esegue la moltiplicazione, assume il seguente aspetto:

```
mov      eax, moltiplicatore        ; eax = moltiplicatore
mul      dword ptr moltiplicando[0] ; edx:eax = eax * moltiplicando[0]
mov      prodotto[0], eax           ; salva la prima DWORD del prodotto
mov      ecx, edx                   ; salva il riporto in ecx

mov      eax, moltiplicatore        ; eax = moltiplicatore
mul      dword ptr moltiplicando[4] ; edx:eax = eax * moltiplicando[4]
add      eax, ecx                   ; somma eax con il riporto precedente
mov      prodotto[4], eax           ; salva la seconda DWORD del prodotto
adc      edx, 0                     ; se CF=1, incrementa il riporto
mov      ecx, edx                   ; salva il riporto in ecx

mov      eax, moltiplicatore        ; eax = moltiplicatore
mul      dword ptr moltiplicando[8] ; edx:eax = eax * moltiplicando[8]
add      eax, ecx                   ; somma eax con il riporto precedente
mov      prodotto[8], eax           ; salva la terza DWORD del risultato
adc      edx, 0                     ; se CF=1, incrementa il riporto
mov      prodotto[12], edx          ; salva la quarta DWORD del risultato
```

L'operatore **DWORD PTR** e' stato inserito solo per rendere piu' chiaro il codice; la sua presenza e' superflua in quanto la variabile **moltiplicando** e' stata definita di tipo **DWORD**, e rende quindi esplicito, l'utilizzo di operandi a **32** bit.
Se ora vogliamo visualizzare il risultato con **writeHex32**, possiamo utilizzare lo stesso metodo illustrato per l'istruzione **ADC**.

L'esempio appena svolto e' molto semplice, grazie al fatto che il moltiplicatore, e' formato da una sola **DWORD**, direttamente gestibile dalla **CPU**; cosa succede se anche il moltiplicatore e' formato da due o piu' **DWORD**?
Anche in un caso del genere, dobbiamo applicare lo stesso metodo che si segue con carta e penna; a tale proposito, supponiamo di voler calcolare:
1CF8h * 3BA6h = 06BFF0D0h

Il moltiplicatore e' formato da **4** cifre, per cui dobbiamo calcolare **4** prodotti con lo stesso metodo illustrato in precedenza; il primo prodotto e':
1CF8h * 6h = 0000ADD0h
Il secondo prodotto e':
1CF8h * Ah = 000121B0h
Il terzo prodotto e':
1CF8h * Bh = 00013EA8h
Il quarto prodotto e':
1CF8h * 3h = 000056E8h

I **4** prodotti cosi' ottenuti, vanno sommati tra loro, ricordando, pero', che:

***** le cifre del primo prodotto devono essere shiftate di **0** posti verso sinistra;
***** le cifre del secondo prodotto devono essere shiftate di **1** posto verso sinistra;
***** le cifre del terzo prodotto devono essere shiftate di **2** posti verso sinistra;
***** le cifre del quarto prodotto devono essere shiftate di **3** posti verso sinistra.

Per evitare che i vari shift verso sinistra, applicati a ciascun prodotto parziale, possano provocare il trabocco di cifre significative, e' importante che gli stessi prodotti parziali, vengano rappresentati con almeno **8** cifre esadecimali (**32** bit); infatti, moltiplicando tra loro due fattori a **4** cifre esadecimali, si ottiene un prodotto che richiede, al massimo, **4+4=8** cifre esadecimali.
In riferimento al nostro esempio, il prodotto parziale massimo che possiamo ottenere e':
`FFFFh * Fh = 000EFFF1h`
Il massimo numero di shift verso sinistra, viene applicato al quarto prodotto parziale, ed e' pari a **3** posti; nel nostro caso, il valore **000EFFF1h**, diventa **EFFF1000h**, senza trabocco da sinistra di cifre significative.

Tornando al nostro esempio, la somma tra i **4** prodotti parziali produce il seguente risultato:

```
0000ADD0h +
00121B00h +
013EA800h +
056E8000h =
-----------
06BFF0D0h
```

Per mettere in pratica la tecnica appena illustrata, osserviamo che, come al solito, se decidiamo di operare sulle singole **DWORD** dei due fattori, ciascuna di queste **DWORD** rappresenta una delle cifre esadecimali dell'esempio appena illustrato; supponiamo allora di voler calcolare:
`2BF5218A9ECB00A3h * 86CDAF9731445EDBh = 1725A100F3199F0751EF6E14C0316571h`
Moltiplicando un numero a **64** bit per un numero a **64** bit, si ottiene un risultato a **64+64=128** bit; nel blocco dati del nostro programma, possiamo inserire allora le seguenti definizioni:

```
moltiplicando   dq    2BF5218A9ECB00A3h
moltiplicatore dq    86CDAF9731445EDBh
parziale1       dd    4 dup (0)
parziale2       dd    4 dup (0)
prodotto        dd    4 dup (0)
```

Applichiamo ora la tecnica che gia' conosciamo, per moltiplicare il **moltiplicando** per ciascuna **DWORD** del **moltiplicatore**, a partire da quella meno significativa; in questo modo, otteniamo **2** prodotti parziali.
Il primo prodotto parziale consiste nel calcolare:
`parziale1 = moltiplicando * (dword ptr moltiplicatore[0])`
In questo modo otteniamo il valore a **96** bit:
`parziale1 = 2BF5218A9ECB00A3h * 31445EDBh = 0875A8D20E3BA0EFC0316571h`
Il secondo prodotto parziale consiste nel calcolare:
`parziale2 = moltiplicando * (dword ptr moltiplicatore[4])`
In questo modo otteniamo il valore a **96** bit:
`parziale2 = 2BF5218A9ECB00A3h * 86CDAF97h = 1725A100EAA3F63543B3CD25h`

I **2** prodotti parziali cosi' ottenuti, vanno sommati tra loro, ricordando, pero', che:

***** le cifre del primo prodotto devono essere shiftate di **0 DWORD** verso sinistra;
***** le cifre del secondo prodotto devono essere shiftate di **1 DWORD** verso sinistra.

Per evitare che i vari shift verso sinistra, applicati a ciascun prodotto parziale, possano provocare il trabocco di cifre significative, e' importante che gli stessi prodotti parziali vengano rappresentati con almeno **32** cifre esadecimali (**128** bit); infatti, moltiplicando tra loro due fattori a **16** cifre esadecimali, si ottiene un prodotto che richiede, al massimo, **16+16=32** cifre esadecimali.
Osserviamo che il prodotto parziale massimo che possiamo ottenere e':
`FFFFFFFFFFFFFFFFh * FFFFFFFFh = 00000000FFFFFFFEFFFFFFFF00000001h`
In riferimento al nostro esempio, il massimo numero di shift verso sinistra, viene applicato al secondo prodotto parziale, ed e' pari a **1 DWORD**; nel nostro caso, il valore **00000000FFFFFFFEFFFFFFFF00000001h**, diventa **FFFFFFFEFFFFFFFF0000000100000000h**, senza trabocco da sinistra di cifre significative.

Tornando al nostro esempio, la somma tra i **2** prodotti parziali produce il seguente risultato:

```
000000000875A8D20E3BA0EFC0316571h +
1725A100EAA3F63543B3CD2500000000h =
-----------------------------------
1725A100F3199F0751EF6E14C0316571h
```

Per shiftare i vari prodotti parziali, possiamo anche fare a meno delle istruzioni di shifting (che verranno illustrate in un altro capitolo); infatti, osservando la struttura della somma illustrata in precedenza, possiamo scrivere il seguente codice:

```
mov      eax, parziale1[0]       ; eax = parziale1[0]
mov      prodotto[0], eax        ; salva la prima DWORD del risultato

mov      eax, parziale1[4]       ; eax = parziale1[4]
add      eax, parziale2[0]       ; eax = parziale1[4] + parziale2[0]
mov      prodotto[4], eax        ; salva la seconda DWORD del risultato

mov      eax, parziale1[8]       ; eax = parziale1[8]
adc      eax, parziale2[4]       ; eax = parziale1[8] + parziale2[4] + CF
mov      prodotto[8], eax        ; salva la terza DWORD del risultato

mov      eax, parziale2[8]       ; eax = parziale2[8]
adc      eax, 0                  ; eax = parziale2[8] + CF
mov      prodotto[12], eax       ; salva la quarta DWORD del risultato
```

In sostanza, la **DWORD** meno significativa del risultato, coincide con la **DWORD** meno significativa di **parziale1**; la **DWORD** piu' significativa del risultato, coincide con la **DWORD** piu' significativa di **parziale2** (sommata ad un eventuale **carry** precedente).

17.9.5 Effetti provocati da MUL sugli operandi e sui flags

In base alle considerazioni esposte in precedenza, risulta che l'esecuzione dell'istruzione **MUL** con operando a **8** bit, modifica il solo registro **AX**; l'esecuzione dell'istruzione **MUL** con operando a **16** bit, modifica i registri **DX** e **AX**. L'esecuzione dell'istruzione **MUL** con operando a **32** bit, modifica i registri **EDX** e **EAX**.

La possibilita' di effettuare una moltiplicazione tra **Reg** e **Reg**, ci permette di scrivere istruzioni del tipo:

```
mul ax
```

Come si puo' facilmente constatare, questa istruzione calcola il **quadrato** di **AX**; se, ad esempio, **AX=4850**, l'istruzione precedente ci fornisce il risultato:

DX:AX = AX * AX = 4850 * 4850 = 23522500 = 4850^2

L'esecuzione dell'istruzione **MUL** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; i campi **PF**, **AF**, **ZF** e **SF**, assumono un valore indeterminato, e non hanno quindi nessun significato.
I campi **CF** e **OF** indicano se, l'esecuzione di **MUL** con operandi a **n** bit, ha prodotto un risultato interamente rappresentabile con soli **n** bit; se il risultato richiede solamente **n** bit, la **CPU** pone **CF=0** e **OF=0**, mentre se il risultato richiede tutti i **2n** bit, la **CPU** pone **CF=1** e **OF=1**.

Poniamo, ad esempio, **AX=2390**, **BX=24**, ed eseguiamo l'istruzione:

```
mul bx
```

In presenza di questa istruzione, la **CPU** calcola:

```
DX:AX = AX * BX = 2390 * 24 = 57360 = 0000E010h
```

Il valore **57360** e' minore di **65536**, ed e' quindi interamente rappresentabile con soli **16** bit; si ottiene, infatti, **DX=0000h** e **AX=E010h**.
Per segnalare questa situazione, la **CPU** pone **CF=0** e **OF=0**.

Poniamo, ad esempio, **AX=4500**, **BX=175**, ed eseguiamo l'istruzione:

```
mul bx
```

In presenza di questa istruzione, la **CPU** calcola:

```
DX:AX = AX * BX = 4500 * 175 = 787500 = 000C042Ch
```

Il valore **787500** e' maggiore di **65535**, e non e' quindi interamente rappresentabile con soli **16** bit; si ottiene, infatti, **DX=000Ch** e **AX=042Ch**.
Per segnalare questa situazione, la **CPU** pone **CF=1** e **OF=1**.

17.10 L'istruzione IMUL

Con il mnemonico **IMUL** si indica l'istruzione **Integer signed MULtiply** (moltiplicazione tra numeri interi con segno); le uniche forme lecite per l'istruzione **IMUL**, sono le seguenti:

```
IMUL     Reg
IMUL     Mem
IMUL     Reg16, Reg16
IMUL     Reg16, Mem16
IMUL     Reg16, Imm
IMUL     Reg32, Reg32
IMUL     Reg32, Mem32
IMUL     Reg32, Imm
IMUL     Reg16, Reg16, Imm8
IMUL     Reg16, Mem16, Imm8
IMUL     Reg32, Reg32, Imm8
IMUL     Reg32, Mem32, Imm8
```

Come si puo' notare, a differenza di quanto accade con **MUL**, l'istruzione **IMUL** prevede diverse combinazioni tra operando **SRC** e operando **DEST**; tali combinazioni, verranno analizzate in dettaglio piu' avanti.

Come abbiamo visto nel capitolo dedicato alla matematica del computer, moltiplicando tra loro due numeri interi con segno a **n** bit in complemento a **2**, otteniamo un risultato che occupa, al massimo, **2n** bit; inoltre, il prodotto che si ottiene con due fattori a **n** bit, e' sempre compreso tra il valore minimo e il valore massimo, rappresentabili con **2n** bit.
Considerando, ad esempio, numeri interi con segno a **8** bit in complemento a **2**, il prodotto minimo che possiamo ottenere e':
`(-128) * (+127) = -16256`
Il valore **-16256** e' nettamente superiore al valore minimo (**-32768**) rappresentabile con **16** bit.
Analogamente, il prodotto massimo che possiamo ottenere e':
`(-128) * (-128) = +16384`
Il valore **+16384** e' nettamente inferiore al valore massimo (**+32767**) rappresentabile con **16** bit.

Come gia' sappiamo, la moltiplicazione provoca un cambiamento di modulo; di conseguenza, **MUL** e **IMUL** forniscono lo stesso risultato solo se i due fattori rappresentano numeri interi positivi, sia nell'insieme dei numeri interi senza segno, sia nell'insieme dei numeri interi con segno.

Se vogliamo conoscere le convenzioni seguite dalla **CPU** per la memorizzazione del risultato della moltiplicazione eseguita con **IMUL**, dobbiamo analizzare tutti i casi che si possono presentare; il risultato fornito da **IMUL**, e' sempre rappresentato, ovviamente, in complemento a **2**.

17.10.1 Istruzione IMUL con un solo operando esplicito

In questo caso, il programmatore specifica, esplicitamente, il solo operando **SRC**, che puo' essere di tipo **Reg** o **Mem**; e' proibito l'uso di operandi di tipo **Imm**. La situazione e' del tutto simile a quella gia' esaminata per l'istruzione **MUL**; infatti, in base all'ampiezza in bit (**8**, **16** o **32**) dell'operando **SRC**, la **CPU** decide se moltiplicare, rispettivamente, per **AL**, per **AX** o per **EAX**.
Anche le convenzioni seguite dalla **CPU** per la memorizzazione del risultato, sono le stesse gia' illustrate per **MUL**; possiamo dire quindi che, se l'operando e' a **8** bit (**SRC8**):
```
AX = AL * SRC8
```
Se l'operando e' a **16** bit (**SRC16**):
```
DX:AX = AX * SRC16
```
Se l'operando e' a **32** bit (**SRC32**):
```
EDX:EAX = EAX * SRC32
```

Come al solito, nei tre casi possibili, la parte piu' significativa del risultato si trova, rispettivamente, in **AH**, in **DX**, in **EDX**.

17.10.2 Istruzione IMUL con due operandi espliciti

In questo caso, il programmatore specifica, esplicitamente, sia l'operando **DEST** (operando piu' a sinistra), sia l'operando **SRC** (operando piu' a destra); l'operando **DEST** puo' essere, esclusivamente, di tipo **Reg16** o **Reg32**. Se l'operando **DEST** e' di tipo **Reg16**, allora l'operando **SRC** puo' essere di tipo **Reg16**, **Mem16** o **Imm**; se l'operando **DEST** e' di tipo **Reg32**, allora l'operando **SRC** puo' essere di tipo **Reg32**, **Mem32** o **Imm**. Un operando **SRC** di tipo **Imm**, viene eventualmente sottoposto all'estensione del bit di segno, sino a raggiungere l'ampiezza in bit di **DEST**.

In pratica, con questa forma dell'istruzione **IMUL**, stiamo indicando alla **CPU**, anche l'operando nel quale deve essere memorizzato il prodotto; di conseguenza, la **CPU** calcola:
```
DEST = DEST * SRC
```
Osserviamo, pero', che l'operando **DEST**, ha la stessa ampiezza in bit dell'operando **SRC**; cio' significa che questa forma dell'istruzione **IMUL**, moltiplica due operandi, **SRC** e **DEST**, entrambi a **n** bit, e memorizza il risultato nell'operando **DEST** a **n** bit. Di conseguenza, puo' capitare che il risultato della moltiplicazione, ecceda i limiti, minimo e massimo, rappresentabili con **n** bit; piu' avanti, vedremo come la **CPU** segnala questa situazione di errore.

17.10.3 Istruzione IMUL con tre operandi espliciti

In questo caso, il programmatore specifica, esplicitamente, l'operando **DEST** (operando piu' a sinistra) nel quale verra' memorizzato il prodotto, l'operando **SRC1** (operando centrale) che rappresenta il primo fattore, e l'operando **SRC2** (operando piu' a destra) che rappresenta il secondo fattore; l'operando **DEST** puo' essere, esclusivamente, di tipo **Reg16** o **Reg32**. L'operando **SRC1** puo' essere di tipo **Reg** o **Mem**, e deve avere la stessa ampiezza in bit di **DEST**; l'operando **SRC2** deve essere, esclusivamente, di tipo **Imm8**, e viene quindi sottoposto all'estensione del bit di segno, sino a raggiungere l'ampiezza in bit di **DEST**.

In pratica, con questa forma dell'istruzione **IMUL**, stiamo indicando alla **CPU**, l'operando che contiene il primo fattore (**SRC1**), l'operando che contiene il secondo fattore (**SRC2**), e anche l'operando nel quale deve essere memorizzato il prodotto tra i due fattori (**DEST**); di conseguenza, la **CPU** calcola:
```
DEST = SRC1 * SRC2
```
Osserviamo, pero', che l'operando **DEST**, ha la stessa ampiezza in bit dell'operando **SRC1**; cio'

significa che questa forma dell'istruzione **IMUL**, moltiplica due operandi, **SRC1** e **SRC2**, entrambi a **n** bit, e memorizza il risultato nell'operando **DEST** a **n** bit. Di conseguenza, puo' capitare che il risultato della moltiplicazione, ecceda i limiti, minimo e massimo, rappresentabili con **n** bit; piu' avanti, vedremo come la **CPU** segnala questa situazione di errore.

17.10.4 Moltiplicazione tra numeri interi con segno di ampiezza arbitraria

Per moltiplicare numeri interi con segno di ampiezza arbitraria, si puo' sfruttare il fatto che, il valore assoluto del prodotto, e' indipendente dal segno dei due fattori; ad esempio:
```
| 98 * 42 | = 4116
```
e:
```
| (-98) * (+42) | = 4116
```

Possiamo servirci allora dello stesso procedimento gia' illustrato per l'istruzione **MUL**; a tale proposito, possiamo suddividere l'algoritmo nelle seguenti fasi:

1) memorizzazione del segno dei due fattori;
2) cambiamento di segno (negazione) di eventuali fattori negativi;
3) esecuzione della moltiplicazione tra numeri interi senza segno (con **MUL**);
4) adeguamento del risultato in base al segno.

In relazione alla fase n. **4**, osserviamo che il segno del prodotto e' dato dalle note regole seguenti:

(+) * (+) = (+) (il prodotto e' un numero positivo);
(+) * (-) = (-) (il prodotto e' un numero negativo);
(-) * (+) = (-) (il prodotto e' un numero negativo);
(-) * (-) = (+) (il prodotto e' un numero positivo).

Se la moltiplicazione deve fornire un prodotto positivo, non dobbiamo apportare nessuna modifica al risultato ottenuto; se, invece, la moltiplicazione deve fornire un prodotto negativo, dobbiamo cambiare di segno (negare) il risultato ottenuto.
Dalle considerazioni appena esposte, risulta che, rispetto all'esempio mostrato per **MUL**, abbiamo bisogno anche di un algoritmo per la negazione dei numeri interi con segno di ampiezza arbitraria; quando avremo a disposizione tutti gli strumenti necessari, analizzeremo un esempio pratico.

17.10.5 Effetti provocati da IMUL sugli operandi e sui flags

L'esecuzione dell'istruzione **IMUL** con un solo operando esplicito a **8**, **16** o **32** bit, provoca la modifica, rispettivamente, dei registri **AX**, **DX:AX** e **EDX:EAX**.
La possibilita' di utilizzare **IMUL** con operando **SRC** di tipo **Reg**, ci permette di scrivere istruzioni del tipo:
```
imul eax
```
L'esecuzione di questa istruzione, memorizza in **EDX:EAX** il **quadrato** (sempre positivo) del numero intero con segno contenuto in **EAX**.

L'esecuzione dell'istruzione **IMUL** con due operandi espliciti, provoca la modifica del solo operando **DEST**, il cui contenuto viene sovrascritto dal risultato della moltiplicazione; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna prestare attenzione ai casi del tipo:
```
imul bx, [bx]
```
Dopo l'esecuzione di questa istruzione, il contenuto del registro puntatore **BX** viene sovrascritto dal risultato della moltiplicazione.

La possibilita' di utilizzare **IMUL** con operandi **SRC** e **DEST** di tipo **Reg**, ci permette di scrivere istruzioni del tipo:

```
imul cx, cx
```

L'esecuzione di questa istruzione, memorizza in **CX** (salvo overflow) il **quadrato** (sempre positivo) del numero intero con segno contenuto in **CX**.

L'esecuzione dell'istruzione **IMUL** con tre operandi espliciti, provoca la modifica del solo operando **DEST**, il cui contenuto viene sovrascritto dal risultato della moltiplicazione; il contenuto degli operandi **SRC1** e **SRC2** rimane inalterato.
Bisogna prestare attenzione ai casi del tipo:

```
imul bx, [bx+si], +35
```

Dopo l'esecuzione di questa istruzione, il contenuto del registro puntatore **BX** viene sovrascritto dal risultato della moltiplicazione.
La possibilita' di utilizzare **IMUL** con operandi **SRC1** e **DEST** di tipo **Reg**, ci permette di scrivere istruzioni del tipo:

```
imul edx, edx, -18
```

L'esecuzione di questa istruzione, memorizza in **EDX** (salvo overflow) il **quadrato**, moltiplicato per
-18, del numero intero con segno contenuto in **EDX**; si noti che in questo tipo di istruzione, la modifica di **DEST** si ripercuote anche su **SRC1**.

Nel caso generale, l'esecuzione dell'istruzione **IMUL** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; i campi **PF**, **AF**, **ZF** e **SF**, assumono un valore indeterminato, e non hanno quindi nessun significato.
I campi **CF** e **OF** assumono, invece, un significato analogo a quello gia' illustrato per l'istruzione **MUL**; questa volta, pero', il programmatore deve prestare grande attenzione al fatto che, la **CPU**, pone **CF=1** e **OF=1** per segnalare una situazione di errore!

a) istruzione IMUL con un solo operando esplicito

Nel caso dell'istruzione **IMUL** con un solo operando esplicito, la situazione e' del tutto simile a quella descritta per **MUL**; il risultato della moltiplicazione tra operandi a **n** bit e' sempre esatto, in quanto viene memorizzato con una ampiezza di **2n** bit.
I campi **CF** e **OF** indicano se, l'esecuzione di **IMUL** con operandi a **n** bit, ha prodotto un risultato interamente rappresentabile con soli **n** bit; se il risultato richiede solamente **n** bit, la **CPU** pone **CF=0** e **OF=0**, mentre se il risultato richiede tutti i **2n** bit, la **CPU** pone **CF=1** e **OF=1**.

Poniamo, ad esempio, **AX=-890**, **BX=24**, ed eseguiamo l'istruzione:

```
imul bx
```

In presenza di questa istruzione, la **CPU** calcola:

```
DX:AX = AX * BX = -890 * 24 = -21360 = 11111111111111111010110010010000b
```

Trascurando i **16** bit piu' significativi del risultato, contenuti in **DX**, otteniamo:

```
AX = 1010110010010000b = 44176
```

Per i numeri interi con segno a **16** bit in complemento a **2**, il valore **44176** rappresenta il numero negativo **-21360**; infatti:

$44176 - 2^{16} = -21360$

Come si puo' notare, se trascuriamo il contenuto di **DX**, otteniamo in **AX** un risultato ugualmente valido; cio' accade in quanto il valore **-21360** e' maggiore di **-32768**, ed e' quindi interamente rappresentabile con soli **16** bit.
In pratica, il numero negativo **-21360** contenuto in **DX:AX**, puo' essere rappresentato con soli **16** bit senza il rischio di perdere il suo bit di segno; per segnalare questa situazione, la **CPU** pone **CF=0** e **OF=0**.

Poniamo, ad esempio, **AX=-4500**, **BX=175**, ed eseguiamo l'istruzione:

```
imul bx
```

In presenza di questa istruzione, la **CPU** calcola:

DX:AX = AX * BX = -4500 * 175 = -787500 = 11111111111100111111101111010100b

Trascurando i **16** bit piu' significativi del risultato, contenuti in **DX**, otteniamo:

AX = 1111101111010100b = 64468

Per i numeri interi con segno a **16** bit in complemento a **2**, il valore **64468** rappresenta il numero negativo **-1068**; infatti:

$64468 - 2^{16} = -1068$

Come si puo' notare, se trascuriamo il contenuto di **DX**, otteniamo in **AX** un risultato privo di senso; cio' accade in quanto il valore **-787500** e' minore di **-32768**, e per la sua rappresentazione richiede quindi piu' di **16** bit.
In pratica, il numero negativo **-787500** contenuto in **DX:AX**, non puo' essere rappresentato con soli **16** bit, perche' si perderebbe il suo bit di segno; per segnalare questa situazione, la **CPU** pone **CF=1** e **OF=1**.

b) istruzione IMUL con due operandi espliciti

Anche nel caso dell'istruzione **IMUL** con due operandi espliciti, i due campi **CF** e **OF** indicano se, l'esecuzione di **IMUL** con operandi a **n** bit, ha prodotto un risultato interamente rappresentabile con soli **n** bit; questa volta, pero', bisogna ricordare che l'istruzione **IMUL**, memorizza il risultato negli **n** bit dell'operando **DEST**!
Se il risultato richiede solamente **n** bit, la **CPU** pone **CF=0** e **OF=0**; in questo caso, l'operando **DEST** a **n** bit, contiene il risultato esatto della moltiplicazione.
Se il risultato richiede piu' di **n** bit, la **CPU** pone **CF=1** e **OF=1**; in questo caso, l'operando **DEST** a **n** bit, contiene solamente gli **n** bit meno significativi del risultato. Tale risultato e' quindi inutilizzabile in quanto ha subito un troncamento di cifre significative; in una situazione di questo genere, non ci resta che ricorrere all'istruzione **IMUL** con un solo operando esplicito (risultato esatto a **2n** bit).

Poniamo, ad esempio, **CX=-890**, **DX=24**, ed eseguiamo l'istruzione:

```
imul cx, dx
```

In presenza di questa istruzione, la **CPU** calcola:

Temp32 = CX * DX = -890 * 24 = -21360 = 11111111111111111010110010010000b

La **CPU** legge i **16** bit meno significativi contenuti nel registro temporaneo **Temp32**, e li copia in **CX**; si ottiene quindi:

CX = 1010110010010000b = 44176

Per i numeri interi con segno a **16** bit in complemento a **2**, il valore **44176** rappresenta il numero negativo **-21360**; infatti:

$44176 - 2^{16} = -21360$

Come si puo' notare, il troncamento effettuato dalla **CPU** ha prodotto un risultato ugualmente valido; cio' accade in quanto il valore **-21360** e' maggiore di **-32768**, ed e' quindi interamente rappresentabile con soli **16** bit.
In pratica, il numero negativo **-21360**, contenuto in **Temp32**, puo' essere rappresentato con soli **16** bit, in quanto il troncamento non provoca la perdita del bit di segno; per segnalare questa situazione, la **CPU** pone **CF=0** e **OF=0**.

Poniamo, ad esempio, **CX=-4500**, **DX=175**, ed eseguiamo l'istruzione:

```
imul cx, dx
```

In presenza di questa istruzione, la **CPU** calcola:

Temp32 = CX * DX = -4500 * 175 = -787500 = 11111111111100111111101111010100b

La **CPU** legge i **16** bit meno significativi contenuti nel registro temporaneo **Temp32**, e li copia in **CX**; si ottiene quindi:

```
CX = 1111101111010100b = 64468
```

Per i numeri interi con segno a **16** bit in complemento a **2**, il valore **64468** rappresenta il numero negativo **-1068**; infatti:

```
64468 - 2^16 = -1068
```

Come si puo' notare, il troncamento effettuato dalla **CPU** ha prodotto un risultato privo di senso; cio' accade in quanto il valore **-787500** e' minore di **-32768**, e per la sua rappresentazione richiede quindi piu' di **16** bit.
In pratica, il numero negativo **-787500**, contenuto in **Temp32**, non puo' essere rappresentato con soli **16** bit, in quanto il troncamento provoca la perdita del bit di segno; per segnalare questa situazione, la **CPU** pone **CF=1** e **OF=1**.

c) istruzione IMUL con tre operandi espliciti

Il caso dell'istruzione **IMUL** con tre operandi espliciti, e' assolutamente identico al precedente caso **b**; i campi **CF** e **OF** indicano se, l'esecuzione di **IMUL** con operandi a **n** bit, ha prodotto un risultato interamente rappresentabile con soli **n** bit.
Se il risultato richiede solamente **n** bit, la **CPU** pone **CF=0** e **OF=0**; in questo caso, l'operando **DEST** a **n** bit, contiene il risultato esatto della moltiplicazione.
Se il risultato richiede piu' di **n** bit, la **CPU** pone **CF=1** e **OF=1**; in questo caso, l'operando **DEST** a **n** bit, contiene solamente gli **n** bit meno significativi del risultato. Tale risultato e' quindi inutilizzabile in quanto ha subito un troncamento di cifre significative; in una situazione di questo genere, non ci resta che ricorrere all'istruzione **IMUL** con un solo operando esplicito (risultato esatto a **2n** bit).

17.11 L'istruzione DIV

Con il mnemonico **DIV** si indica l'istruzione **unsigned DIVision of AL/AX/EAX** (divisione di **AL/AX/EAX** per un numero intero senza segno); le uniche forme lecite per l'istruzione **DIV**, sono le seguenti:

```
DIV      Reg
DIV      Mem
```

Come si puo' notare, il programmatore deve specificare, esplicitamente, il solo operando **SRC**, che rappresenta il **divisore**; il **dividendo** e', implicitamente, **AL**, **AX** o **EAX**. A seconda dell'ampiezza in bit (**8**, **16** o **32**) dell'operando **SRC**, la **CPU** decide se il **dividendo** deve essere, rispettivamente, **AL**, **AX** o **EAX**; e' proibito l'uso di un operando **SRC** di tipo **Imm**.

Come abbiamo visto nel capitolo dedicato alla matematica del computer, dividendo tra loro due numeri interi senza segno da **n** bit ciascuno, otteniamo un risultato (**quoziente**) minore o uguale al **dividendo**; inoltre, il **resto** e' sempre inferiore al **divisore**. Da queste considerazioni, segue che la **CPU**, per memorizzare il **quoziente** e il **resto** della divisione, utilizza due registri a **n** bit.
La divisione si svolge nell'insieme dei numeri interi senza segno, e produce quindi sempre un **quoziente** intero, con troncamento della eventuale parte frazionaria; di conseguenza, il **resto** e' nullo solo quando il **dividendo** e' un multiplo intero del **divisore**.

Nel capitolo dedicato alle reti combinatorie, abbiamo anche visto che l'algoritmo per l'esecuzione di una divisione intera, provoca un cambiamento di modulo; proprio per questo motivo, la **CPU** dispone di una istruzione che divide numeri interi senza segno (**DIV**), e di una istruzione che divide numeri interi con segno (**IDIV**).

Se vogliamo conoscere le convenzioni seguite dalla **CPU** per l'esecuzione della divisione con **DIV**,

dobbiamo analizzare i tre casi fondamentali che si possono presentare; nel seguito del capitolo, indichiamo l'operatore **divisione** con il simbolo '/' (slash), mentre il simbolo '**:**', viene utilizzato, come al solito, per separare una coppia di valori disposti secondo la convenzione **little endian** (parte piu' significativa a sinistra e parte meno significativa a destra).

17.11.1 Divisione tra numeri interi senza segno a 8 bit

Vogliamo calcolare:
```
240 / 27 (Q = 8, R = 24)
```
Poniamo, **AL=240**, **BH=27**, ed eseguiamo l'istruzione:
```
div bh
```
In presenza di questa istruzione, la **CPU** utilizza implicitamente il registro **AL**, e lo unisce al registro **AH** in modo che il **dividendo** sia rappresentato dalla coppia **AH:AL**; a questo punto, la **CPU** calcola:
```
AH:AL = AH:AL / BH = 240 / 27 = 24:8 = 18h:08h
```
Come si puo' notare, il **quoziente** a **8** bit viene memorizzato in **AL**, mentre il **resto** a **8** bit, viene memorizzato in **AH**.

E' importantissimo osservare che la **CPU** utilizza **AH:AL** come **dividendo**; di conseguenza, affinche' **DIV** fornisca un risultato valido, e' necessario che il programmatore estenda in **AH** il segno del **dividendo** stesso, contenuto in **AL**!
Ricordando che stiamo operando nell'insieme dei numeri interi senza segno, prima di eseguire **DIV** con operandi a **8** bit dobbiamo porre **AH=0** (**zero extension** del valore contenuto in **AL**); se dimentichiamo di azzerare **AH**, l'istruzione **DIV** produce un risultato che, in certi casi (che verranno analizzati nel seguito), provoca l'interruzione del programma a causa di un **overflow di divisione**!

17.11.2 Divisione tra numeri interi senza segno a 16 bit

Vogliamo calcolare:
```
49750 / 832 (Q = 59, R = 662)
```
Poniamo, **AX=49750**, e supponiamo che **ES:(BX+SI)** punti in memoria ad una **WORD** che vale **832**; eseguiamo ora l'istruzione:
```
div word ptr es:[bx+si]
```
L'operatore **WORD PTR** e' necessario per indicare all'assembler che gli operandi sono a **16** bit; il segment override e' necessario perche' **ES** non e' il registro di segmento naturale per i dati.
In presenza di questa istruzione, la **CPU** utilizza implicitamente il registro **AX**, e lo unisce al registro **DX** in modo che il **dividendo** sia rappresentato dalla coppia **DX:AX**; a questo punto, la **CPU** calcola:
```
DX:AX = DX:AX / ES:[BX+SI] = 49750 / 832 = 662:59 = 0296h:003Bh
```
Come si puo' notare, il **quoziente** a **16** bit viene memorizzato in **AX**, mentre il **resto** a **16** bit, viene memorizzato in **DX**.

E' importantissimo osservare che la **CPU** utilizza **DX:AX** come **dividendo**; di conseguenza, affinche' **DIV** fornisca un risultato valido, e' necessario che il programmatore estenda in **DX** il segno del **dividendo** stesso, contenuto in **AX**!
Ricordando che stiamo operando nell'insieme dei numeri interi senza segno, prima di eseguire **DIV** con operandi a **16** bit dobbiamo porre **DX=0** (**zero extension** del valore contenuto in **AX**); se dimentichiamo di azzerare **DX**, l'istruzione **DIV** produce un risultato che, in certi casi (che verranno analizzati nel seguito), provoca l'interruzione del programma a causa di un **overflow di divisione**!

17.11.3 Divisione tra numeri interi senza segno a 32 bit

Vogliamo calcolare:
```
2997860228 / 384000 (Q = 7806, R = 356228)
```
Poniamo, **EAX=2997860228**, **ECX=384000**, ed eseguiamo l'istruzione:
```
div ecx
```
In presenza di questa istruzione, la **CPU** utilizza implicitamente il registro **EAX**, e lo unisce al registro **EDX** in modo che il **dividendo** sia rappresentato dalla coppia **EDX:EAX**; a questo punto, la **CPU** calcola:
```
EDX:EAX = EDX:EAX / ECX = 2997860228 / 384000 = 356228:7806 =
00056F84h:00001E7Eh
```
Come si puo' notare, il **quoziente** a **32** bit viene memorizzato in **EAX**, mentre il **resto** a **32** bit, viene memorizzato in **EDX**.

E' importantissimo osservare che la **CPU** utilizza **EDX:EAX** come **dividendo**; di conseguenza, affinche' **DIV** fornisca un risultato valido, e' necessario che il programmatore estenda in **EDX** il segno del **dividendo** stesso, contenuto in **EAX**!
Ricordando che stiamo operando nell'insieme dei numeri interi senza segno, prima di eseguire **DIV** con operandi a **32** bit dobbiamo porre **EDX=0** (**zero extension** del valore contenuto in **EAX**); se dimentichiamo di azzerare **EDX**, l'istruzione **DIV** produce un risultato che, in certi casi (che verranno analizzati nel seguito), provoca l'interruzione del programma a causa di un **overflow di divisione**!

17.11.4 Divisione tra numeri interi senza segno di ampiezza arbitraria

Come abbiamo appena visto, nell'eseguire l'istruzione **DIV** con operandi a **8**, **16** o **32** bit, la **CPU** presuppone che il **dividendo** si trovi, rispettivamente, in **AH:AL**, in **DX:AX** o in **EDX:EAX**; come mai la **CPU** richiede che il **dividendo** venga rappresentato con il doppio dei bit necessari?
Il perche' di questo comportamento e' legato ad una geniale trovata dei progettisti delle **CPU**; grazie a questa trovata, e' possibile eseguire con relativa facilita', divisioni tra numeri interi di ampiezza arbitraria.
Vediamo subito un esempio pratico che chiarisce questo importante aspetto.

Per dividere numeri interi senza segno di ampiezza arbitraria, possiamo servirci di un algoritmo del tutto simile a quello che si utilizza con carta e penna; per analizzare tale algoritmo, supponiamo di voler calcolare:
```
9AB6h / Ch (Q = 0CE4h, R = 0006h)
```

Come sappiamo, la divisione inizia con la cifra piu' significativa del **dividendo**; nel nostro caso quindi, il primo quoziente parziale da calcolare e':
```
9h / Ch (Q3 = 0h, R3 = 9h)
```
Il quoziente parziale indicato con **Q3** rappresenta la cifra piu' significativa del **quoziente** finale; al resto parziale **R3** dobbiamo affiancare la seconda cifra (da sinistra) del **dividendo**, in modo da ottenere **9Ah**.

Il secondo quoziente parziale da calcolare e':
```
9Ah / Ch (Q2 = Ch, R2 = Ah)
```
Il quoziente parziale indicato con **Q2** rappresenta la seconda cifra (da sinistra) del **quoziente** finale; al resto parziale **R2** dobbiamo affiancare la terza cifra (da sinistra) del **dividendo**, in modo da ottenere **ABh**.

Il terzo quoziente parziale da calcolare e':
```
ABh / Ch (Q1 = Eh, R1 = 3h)
```

Il quoziente parziale indicato con **Q1** rappresenta la terza cifra (da sinistra) del **quoziente** finale; al resto parziale **R1** dobbiamo affiancare la quarta cifra (da sinistra) del **dividendo**, in modo da ottenere **36h**.

Il quarto e ultimo quoziente parziale da calcolare e':
`36h / Ch (Q0 = 4h, R0 = 6h)`
Il quoziente parziale indicato con **Q0** rappresenta la cifra meno significativa del **quoziente** finale; il resto parziale **R0** rappresenta il **resto** finale della divisione.

Unendo i **4** quozienti parziali otteniamo il **quoziente** finale **0CE4h**; inoltre, abbiamo appena visto che l'ultimo resto parziale **R0**, ci fornisce il **resto** finale **6h**.

Come si puo' notare, la prima divisione consiste nel dividere un numero a **1** cifra esadecimale, per un numero a **1** cifra esadecimale; una tale divisione produce sempre un **quoziente** a **1** cifra esadecimale e, ovviamente, un **resto** a **1** cifra esadecimale.
A partire dalla seconda divisione, dobbiamo dividere un numero a **2** cifre esadecimali, per un numero a **1** cifra esadecimale; una tale divisione puo' produrre un **quoziente** a **2** cifre esadecimali?
La risposta e' no!
Per dimostrarlo, osserviamo innanzi tutto che il **divisore** e' formato da **1** sola cifra esadecimale, per cui e' sempre compreso tra **1h** e **Fh**; di conseguenza, anche il **resto**, dovendo essere minore del **divisore**, e' formato sempre da **1** sola cifra esadecimale.
Osserviamo ora che affiancando al resto **Ri** una qualunque cifra **Mj** del **dividendo**, otteniamo un numero a due cifre esadecimali, rappresentato da:
`RiMj = (Ri * 10h) + Mj`
(con la cifra **Mj** che e' sempre compresa tra **0h** e **Fh**).
Ad esempio, se **Ri=3h** e **Mj=Bh**, si ha:
`RiMj = 3Bh = (3h * 10h) + Bh`

A questo punto, possiamo sottoporre **RiMj** alle seguenti elaborazioni:

```
RiMj = (Ri * 10h) + Mj <= (Ri * 10h) + Fh =
     = (Ri * (Fh + 1h)) + Fh =
     = (Ri * Fh) + (Ri * 1h) + Fh =
     = (Ri * Fh) + Ri + Fh =
     = ((Ri + 1h) * Fh) + Ri
```

In sostanza, **RiMj** e' sicuramente minore o uguale a **((Ri + 1) * Fh) + Ri**.
Per calcolare un nuovo quoziente parziale, dobbiamo dividere **RiMj** per il **divisore**, che indichiamo con **N**; di conseguenza, possiamo dire che:

```
RiMj      ((Ri + 1h) * Fh) + Ri
---- <= -----------------------
 N                 N
```

Assegnando ora a **N** un qualunque valore compreso tra **1h** e **Fh**, possiamo constatare che il secondo membro della precedente disequazione, e' sempre minore o uguale a **Fh**!
Ad esempio, se **N=1h**, allora **Ri** non puo' superare **0h**; di conseguenza, il quoziente parziale massimo che possiamo ottenere e':
`(((Ri + 1h) * Fh) + Ri) / N = (((0h + 1h) * Fh) + 0h) / 1h = Fh / 1h = Fh` (con resto max. **0h**).
Se **N=2h**, allora **Ri** non puo' superare **1h**; di conseguenza, il quoziente parziale massimo che possiamo ottenere e':
`(((Ri + 1h) * Fh) + Ri) / N = (((1h + 1h) * Fh) + 1h) / 2h = 1Fh / 2h = Fh` (con

resto max. **1h**).
Se **N=3h**, allora **Ri** non puo' superare **2h**; di conseguenza, il quoziente parziale massimo che possiamo ottenere e':
`(((Ri + 1h) * Fh) + Ri) / N = (((2h + 1h) * Fh) + 2h) / 3h = 2Fh / 3h = Fh` (con resto max. **2h**).
E cosi' via, sino ad arrivare a **N=Fh**, con **Ri** che non puo' quindi superare **Eh**; di conseguenza, il quoziente parziale massimo che possiamo ottenere e':
`(((Ri + 1h) * Fh) + Ri) / N = (((Eh + 1h) * Fh) + Eh) / Fh = EFh / Fh = Fh` (con resto max. **Eh**).

Risulta quindi che, a maggior ragione, il generico quoziente parziale **(RiMj / N)**, non puo' mai superare il valore **Fh**!

E' importante ribadire che, nel calcolo del primo quoziente parziale, il **dividendo** deve essere formato da **1** sola cifra esadecimale; in caso contrario, puo' succedere che, ad esempio:
`EEh / 4h (Q = 3Bh, R = 2h)`
In generale, se otteniamo un quoziente parziale maggiore di **Fh**, vuol dire che abbiamo commesso qualche errore nello svolgimento dei calcoli; in particolare, se **RiMj** e' maggiore di zero, e il **divisore** (**N**) vale zero, si ottiene un quoziente parziale infinitamente grande (divisione per zero)!

Passiamo ora alla implementazione pratica dell'algoritmo per la divisione tra numeri interi senza segno di ampiezza arbitraria; supponendo di avere a disposizione una **CPU 80386** o superiore, ci serviremo dell'istruzione **DIV** con operandi di tipo **DWORD**.
In tal caso, ogni **DWORD** equivale ad una delle cifre esadecimali dell'esempio che abbiamo svolto in precedenza; in particolare, osserviamo che nell'eseguire l'istruzione **DIV**, la **CPU** si serve, implicitamente, della coppia di registri **EDX:EAX**, e ci restituisce i vari quozienti parziali in **EAX** e i vari resti parziali in **EDX**.
Nel calcolare il primo quoziente parziale, dobbiamo caricare in **EAX** la **DWORD** piu' significativa del **dividendo**, e dobbiamo porre, come sappiamo, **EDX=0** (questo passo e' fondamentale); dopo l'esecuzione dell'istruzione **DIV** (tra **EDX:EAX** e il **divisore**) otteniamo, in **EAX** la **DWORD** piu' significativa del **quoziente** finale, e in **EDX** il primo resto parziale.
La **DWORD** contenuta in **EAX** deve essere salvata in memoria, in quanto, nello stesso registro **EAX**, dobbiamo caricare la seconda **DWORD** (da sinistra) del **dividendo**; grazie all'espediente escogitato dai progettisti delle **CPU**, questa nuova **DWORD** caricata in **EAX**, va' quindi ad affiancarsi al contenuto del registro **EDX**, che rappresenta proprio il valore di cui avevamo bisogno, e cioe', il resto parziale della precedente divisione!
La fase successiva consiste quindi nel dividere **EDX:EAX** per il **divisore**; in base a quanto abbiamo visto nel precedente esempio, una tale divisione produce sempre un quoziente parziale non superiore a **FFFFFFFFh**. Per rendercene conto, non dobbiamo fare altro che ripetere la precedente dimostrazione; a tale proposito, dobbiamo tenere presente che, questa volta, il **divisore** non puo' superare **FFFFFFFFh**, il **resto** e' sempre inferiore al divisore, e inoltre:
`RiMj = (Ri * 100000000h) + Mj`

Sempre in analogia con l'esempio svolto in precedenza, possiamo dire che l'ultimo resto parziale che otteniamo, rappresenta anche il **resto** finale della divisione.

A questo punto possiamo procedere con un esempio pratico; a tale proposito, supponiamo di voler calcolare:
`8CB1AF203DA7B9867B04FFD2h / 35D6CE04h (Q = 000000029CFCDB3A3CEA5589h, R = 19016BAEh)`

Dividendo un numero a **96** bit per un numero a **32** bit, si ottiene un **quoziente** che richiede, al massimo, **96** bit, e un **resto** che richiede, al massimo, **32** bit; nel blocco dati del nostro programma, possiamo inserire allora le seguenti definizioni:

```
dividendo   dd    7B04FFD2h, 3DA7B986h, 8CB1AF20h
divisore    dd    35D6CE04h
quoziente   dd    3 dup (0)
resto       dd    0
```

Il codice che esegue la divisione, assume il seguente aspetto:

```
mov      edx, 0                  ; edx = 0
mov      eax, dividendo[8]       ; eax = dividendo[8]
div      divisore                ; edx:eax = edx:eax / divisore
mov      quoziente[8], eax       ; salva la prima DWORD del quoziente

mov      eax, dividendo[4]       ; eax = dividendo[4]
div      divisore                ; edx:eax = edx:eax / divisore
mov      quoziente[4], eax       ; salva la seconda DWORD del quoziente

mov      eax, dividendo[0]       ; eax = dividendo[0]
div      divisore                ; edx:eax = edx:eax / divisore
mov      quoziente[0], eax       ; salva la terza DWORD del quoziente
mov      resto, edx              ; salva il resto finale
```

Se ora vogliamo visualizzare il risultato con **writeHex32**, possiamo utilizzare lo stesso metodo illustrato per l'istruzione **ADC**.

L'esempio appena svolto e' molto semplice, grazie al fatto che il **divisore**, e' formato da una sola **DWORD**, direttamente gestibile dalla **CPU**; cosa succede se anche il **divisore** e' formato da due o piu' **DWORD**?
In un caso del genere, la situazione diventa piu' impegnativa; una soluzione puo' essere quella di simulare il comportamento della rete logica illustrata nella Figura 21 del Capitolo 6. Tale soluzione, comporta l'uso di un algoritmo che occupa parecchio spazio, e necessita anche del procedimento per lo shifting di un numero intero di ampiezza arbitraria; quando avremo a disposizione tutti gli strumenti necessari (istruzioni per i loop e per lo shifting, procedure, etc), analizzeremo un esempio pratico.

17.11.5 Effetti provocati da DIV sugli operandi e sui flags

In base alle considerazioni esposte in precedenza, risulta che l'esecuzione dell'istruzione **DIV** con operando a **8** bit, modifica i registri **AH** e **AL**; l'esecuzione dell'istruzione **DIV** con operando a **16** bit, modifica i registri **DX** e **AX**. L'esecuzione dell'istruzione **DIV** con operando a **32** bit, modifica i registri **EDX** e **EAX**.

La possibilita' di effettuare una divisione tra **Reg** e **Reg**, ci permette di scrivere istruzioni del tipo:
`div ax`
Come si puo' facilmente constatare, l'esecuzione di questa istruzione produce **DX=0** e **AX=1**; naturalmente, e' importante che prima dell'esecuzione di **DIV**, si ponga **DX=0** (inoltre, **AX** deve essere diverso da zero).

L'esecuzione dell'istruzione **DIV** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; tutti questi **6** campi, assumono un valore indeterminato, e non hanno quindi nessun significato.

Ci si puo' chiedere allora come faccia la **CPU** a segnalare eventuali situazioni di errore; per chiarire questo aspetto, dobbiamo analizzare in dettaglio i casi che possono portare l'istruzione **DIV** a provocare un **overflow di divisione**.

a) dividendo troppo grande

Affinche' l'istruzione **DIV** non provochi un **overflow di divisione**, devono verificarsi le seguenti condizioni:

***** nella divisione tra numeri interi senza segno a **8** bit, si deve ottenere un **quoziente** compreso tra **0** e **255**;
***** nella divisione tra numeri interi senza segno a **16** bit, si deve ottenere un **quoziente** compreso tra **0** e **65535**;
***** nella divisione tra numeri interi senza segno a **32** bit, si deve ottenere un **quoziente** compreso tra **0** e **4294967295**.

Se la **CPU** si accorge che queste condizioni non sono verificate, assume che si sia creata una situazione di errore; in tal caso, la **CPU**, anziche' servirsi di un apposito flag, genera (in modalita' reale) una **INT 00h**. Come e' stato spiegato nel Capitolo 14, il vettore di interruzione n. **00h** e' associato ad una **ISR** che segnala un cosiddetto **overflow di divisione**; generalmente, tale **ISR** termina il programma in esecuzione, e mostra il messaggio:

```
Overflow di divisione
```

Come molti avranno intuito, una situazione del genere si verifica, ad esempio, quando, prima di eseguire **DIV** con operandi a **16** bit, dimentichiamo di azzerare **DX**; per dimostrarlo, supponiamo che in **DX** sia presente il valore casuale **842** (**034Ah**).
Poniamo ora **AX=46528** (**B5C0h**), **BX=26**, ed eseguiamo l'istruzione:

```
div bx
```

In presenza di questa istruzione, la **CPU** divide **DX:AX** per **BX**; a causa, pero', della nostra dimenticanza, la coppia **DX:AX** contiene il valore:

```
DX:AX = 034AB5C0h = 55227840
```

Di conseguenza, la divisione dovrebbe fornire il risultato:

```
55227840 / 26 (Q = 2124147, R = 18)
```

E' evidente che il valore **2124147** richiede piu' di **16** bit, e non puo' essere quindi inserito in **AX**; in un caso del genere, la **CPU** rileva la situazione di errore, e genera una **INT 00h**.

Supponiamo, invece, di avere **DX=842** (**034Ah**), **AX=46528** (**B5C0h**), **BX=4126**, ed eseguiamo l'istruzione:

```
div bx
```

In presenza di questa istruzione, la **CPU** divide **DX:AX** per **BX**; la coppia **DX:AX** contiene il valore:

```
DX:AX = 034AB5C0h = 55227840
```

Di conseguenza, la divisione dovrebbe fornire il risultato:

```
55227840 / 4126 (Q = 13385, R = 1330)
```

Questa volta, il valore **13385** puo' essere inserito nei **16** bit di **AX**; in un caso del genere, la **CPU** non segnala nessun errore. Se la nostra intenzione era quella di dividere **55227840** per **4126**, otteniamo un **quoziente** e un **resto** perfettamente validi; se, pero', la nostra intenzione era quella di dividere **46528** per **4126**, otteniamo un risultato privo di senso!

b) divisore uguale a zero

Come sappiamo dalla matematica, dividendo per zero un numero non nullo, otteniamo un **quoziente** infinitamente grande; tale **quoziente**, non puo' essere quindi inserito in nessun registro

della **CPU**.
Anche in questo caso, ci troviamo in una situazione di errore che viene gestita dalla **CPU** attraverso la generazione di una **INT 00h**.
Nota importante
Se abbiamo a disposizione un vero ambiente **DOS**, o la **DOS Box** di **Windows** (compreso l'emulatore **DOS** di **Windows XP**), possiamo verificare direttamente il comportamento della **ISR** associata alla **INT 00h**; a tale proposito, dal prompt del **DOS** dobbiamo impartire il comando:

```
debug
```

Dall'interno del debugger, possiamo richiedere il **dump** della parte iniziale della **RAM** (che contiene i vettori di interruzione); se, ad esempio, vogliamo visualizzare i primi **8** vettori di interruzione (cioe', i primi **8*4=32=20h BYTE** della **RAM**, dal n. **0** al numero **1Fh**), dobbiamo impartire il comando:

```
-d 0000:0000 001f
```

In questo modo, si ottiene il seguente output:

```
-d 0000:0000 001f
0000:0000  68 10 A7 00 8B 01 70 00-16 00 9A 03 8B 01 70 00  h.....p.......p.
0000:0010  8B 01 70 00 B9 06 10 02-40 07 10 02 FF 03 10 02  ..p.....@.......
```

Quest'output si riferisce ad un **PC** con **SO Windows XP**; e' importante tenere presente che queste informazioni, possono variare da computer a computer.
Ricordando che **debug** visualizza il contenuto della memoria, da sinistra verso destra, possiamo dire che la **ISR** associata alla **INT 00h**, si trova all'indirizzo logico **00A7h:1068h**; per eseguire direttamente il codice di questa **ISR**, possiamo impartire il comando **Go** in questo modo:

```
-g = 00a7:1068
```

Le **CPU 80386** e superiori, notificano eventuali situazioni di errore anche attraverso appositi segnali chiamati **eccezioni**; in particolare, in presenza di un **overflow di divisione**, le **CPU 80386** e superiori generano una **eccezione di divisione**, indicata simbolicamente con **#DE** (**divide error**).
I **SO** come **Windows**, **Linux**, etc, possono intercettare quindi queste eccezioni, mostrando apposite finestre che informano l'utente su cio' che e' accaduto; nel caso, ad esempio, di **Windows 98**, in presenza di un **overflow di divisione**, viene mostrata una finestra che ci informa del fatto che il programma in esecuzione ha compiuto una operazione non valida e verra' quindi terminato.

Si raccomanda vivamente di non utilizzare il comando **Go** di **debug**, per eseguire a caso le **ISR** associate ai vari vettori di interruzione; se il programmatore non ha le idee chiare su cio' che sta facendo, puo' provocare un crash che richiede il riavvio del computer!

17.12 L'istruzione IDIV

Con il mnemonico **IDIV** si indica l'istruzione **Integer signed DIVision of AL/AX/EAX** (divisione di **AL/AX/EAX** per un numero intero con segno); le uniche forme lecite per l'istruzione **IDIV**, sono le seguenti:

```
IDIV     Reg
IDIV     Mem
```

Come si puo' notare, il programmatore deve specificare, esplicitamente, il solo operando **SRC**, che rappresenta il **divisore**; il **dividendo** e', implicitamente, **AL**, **AX** o **EAX**. A seconda dell'ampiezza in bit (**8**, **16** o **32**) dell'operando **SRC**, la **CPU** decide se il **dividendo** deve essere, rispettivamente, **AL**, **AX** o **EAX**; e' proibito l'uso di un operando **SRC** di tipo **Imm**.

Dalla matematica sappiamo che, dividendo tra loro due numeri interi con segno da **n** bit ciascuno, otteniamo un **resto** che, in valore assoluto, e' sempre minore del valore assoluto del **divisore**; di conseguenza, il **resto** e' sempre rappresentabile con soli **n** bit, senza il rischio di perdere cifre significative.
Per il **quoziente**, la situazione appare piu' delicata; nel caso, ad esempio, dei numeri interi con segno a **8** bit in complemento a **2**, la **CPU** si aspetta un **quoziente** compreso tra **-128** e **+127**, rappresentabile quindi con soli **8** bit.
Osserviamo pero' che, il **quoziente** minimo che possiamo ottenere e':
`(-128) / (+1) = -128`
Questo valore puo' essere rappresentato con soli **8** bit.
Il **quoziente** massimo che possiamo ottenere e':
`(-128) / (-1) = +128`
Questo valore non puo' essere rappresentato con soli **8** bit!

Ogni volta che la **CPU** ottiene un **quoziente** che non rientra tra i limiti minimo e massimo consentiti, genera un segnale di errore attraverso i metodi gia' illustrati per l'istruzione **DIV**!

Come gia' sappiamo, la divisione provoca un cambiamento di modulo; di conseguenza, **DIV** e **IDIV** forniscono lo stesso risultato solo se il **dividendo** e il **divisore** rappresentano numeri interi positivi, sia nell'insieme dei numeri interi senza segno, sia nell'insieme dei numeri interi con segno.

Se vogliamo conoscere le convenzioni seguite dalla **CPU** per l'esecuzione della divisione con **IDIV**, dobbiamo analizzare i tre casi fondamentali che si possono presentare; il **quoziente** e il **resto** calcolati da **IDIV**, sono sempre rappresentati, ovviamente, in complemento a **2**.

17.12.1 Divisione tra numeri interi con segno a 8 bit

Vogliamo calcolare:
`(-115) / (+18) (Q = -6, R = -7)`
Poniamo, **AL=-115**, **CL=+18**, ed eseguiamo l'istruzione:
`idiv cl`
In presenza di questa istruzione, la **CPU** utilizza implicitamente il registro **AL**, e lo unisce al registro **AH** in modo che il **dividendo** sia rappresentato dalla coppia **AH:AL**; a questo punto, la **CPU** calcola:
`AH:AL = AH:AL / CL = (-115) / (+18) = (-7):(-6) = F9h:FAh`
Come si puo' notare, il **quoziente** a **8** bit viene memorizzato in **AL**, mentre il **resto** a **8** bit viene memorizzato in **AH**.

E' importantissimo osservare che la **CPU** utilizza **AH:AL** come **dividendo**; di conseguenza, affinche' **IDIV** fornisca un risultato valido, e' necessario che il programmatore estenda in **AH** il segno del **dividendo** stesso, contenuto in **AL**!
Ricordando che stiamo operando nell'insieme dei numeri interi con segno, prima di eseguire **IDIV** con operandi a **8** bit dobbiamo caricare il **dividendo** in **AL**, ed eseguire poi l'istruzione:
```
cbw
```
(**sign extension** del valore contenuto in **AL**).
Come si puo' facilmente intuire, non e' certo casuale il fatto che **CBW** utilizzi **AL** come operando **SRC**, e **AH:AL** come operando **DEST**.
Se dimentichiamo di estendere in **AH** il segno del valore contenuto in **AL**, l'istruzione **IDIV** produce un risultato privo di senso che, in certi casi (che verranno analizzati nel seguito), provoca l'interruzione del programma a causa di un **overflow di divisione**!

17.12.2 Divisione tra numeri interi con segno a 16 bit

Vogliamo calcolare:
```
(-16924) / (-696) (Q = +24, R = -220)
```
Poniamo, **AX=-16924**, e supponiamo che **CS:(BX+DI+002Fh)** punti in memoria ad una **WORD** che vale **-696**; eseguiamo ora l'istruzione:
```
idiv word ptr cs:[bx+di+002Fh]
```
L'operatore **WORD PTR** e' necessario per indicare all'assembler che gli operandi sono a **16** bit; il segment override e' necessario perche' **CS** non e' il registro di segmento naturale per i dati.
In presenza di questa istruzione, la **CPU** utilizza implicitamente il registro **AX**, e lo unisce al registro **DX** in modo che il **dividendo** sia rappresentato dalla coppia **DX:AX**; a questo punto, la **CPU** calcola:
```
DX:AX = DX:AX / CS:[BX+DI+002Fh] = (-16924) / (-696) = (-220):(+24) =
FF24h:0018h
```
Come si puo' notare, il **quoziente** a **16** bit viene memorizzato in **AX**, mentre il **resto** a **16** bit viene memorizzato in **DX**.

E' importantissimo osservare che la **CPU** utilizza **DX:AX** come **dividendo**; di conseguenza, affinche' **IDIV** fornisca un risultato valido, e' necessario che il programmatore estenda in **DX** il segno del **dividendo** stesso, contenuto in **AX**!
Ricordando che stiamo operando nell'insieme dei numeri interi con segno, prima di eseguire **IDIV** con operandi a **16** bit dobbiamo caricare il **dividendo** in **AX**, ed eseguire poi l'istruzione:
```
cwd
```
(**sign extension** del valore contenuto in **AX**).
Come si puo' facilmente intuire, non e' certo casuale il fatto che **CWD** utilizzi **AX** come operando **SRC**, e **DX:AX** come operando **DEST**.
Se dimentichiamo di estendere in **DX** il segno del valore contenuto in **AX**, l'istruzione **IDIV** produce un risultato privo di senso che, in certi casi (che verranno analizzati nel seguito), provoca l'interruzione del programma a causa di un **overflow di divisione**!

17.12.3 Divisione tra numeri interi con segno a 32 bit

Vogliamo calcolare:
```
(+2115967835) / (+89736) (Q = +23579, R = +82691)
```
Poniamo, **EAX=+2115967835**, **ECX=+89736**, ed eseguiamo l'istruzione:
```
idiv ecx
```
In presenza di questa istruzione, la **CPU** utilizza implicitamente il registro **EAX**, e lo unisce al registro **EDX** in modo che il **dividendo** sia rappresentato dalla coppia **EDX:EAX**; a questo punto, la **CPU** calcola:
```
EDX:EAX = EDX:EAX / ECX = (+2115967835) / (+89736) = (+82691):(+23579) =
```

```
00014303h:00005C1Bh
```
Come si puo' notare, il **quoziente** a **32** bit viene memorizzato in **EAX**, mentre il **resto** a **32** bit viene memorizzato in **EDX**.

E' importantissimo osservare che la **CPU** utilizza **EDX:EAX** come **dividendo**; di conseguenza, affinche' **IDIV** fornisca un risultato valido, e' necessario che il programmatore estenda in **EDX** il segno del **dividendo** stesso, contenuto in **EAX**!
Ricordando che stiamo operando nell'insieme dei numeri interi con segno, prima di eseguire **IDIV** con operandi a **32** bit dobbiamo caricare il **dividendo** in **EAX**, ed eseguire poi l'istruzione:
```
cdq
```
(**sign extension** del valore contenuto in **EAX**).
Come si puo' facilmente intuire, non e' certo casuale il fatto che **CDQ** utilizzi **EAX** come operando **SRC**, e **EDX:EAX** come operando **DEST**.
Se dimentichiamo di estendere in **EDX** il segno del valore contenuto in **EAX**, l'istruzione **IDIV** produce un risultato privo di senso che, in certi casi (che verranno analizzati nel seguito), provoca l'interruzione del programma a causa di un **overflow di divisione**!
Nota importante
Attenzione a non commettere l'errore di applicare le istruzioni **CBW**, **CWD**, **CWDE** e **CDQ**, ai numeri interi senza segno; in caso contrario, si ottengono risultati privi di senso.
Applichiamo, ad esempio, l'istruzione **CBW** al numero intero senza segno a **8** bit **128** (**10000000b**), contenuto in **AL**; l'istruzione **CBW** crede di avere a che fare con la rappresentazione a **8** bit in complemento a **2**, del numero negativo **-128**. Il bit piu' significativo (**1**) di **AL** viene copiato in tutti gli **8** bit di **AH**, e si ottiene:
AH:AL = 1111111110000000b = 65408
Come si puo' notare, abbiamo ottenuto la rappresentazione a **16** bit in complemento a **2**, del numero intero negativo **-128**; la nostra intenzione, invece, era quella di ottenere **0000000010000000b**, cioe' la rappresentazione a **16** bit del numero intero senza segno **128**!

17.12.4 Divisione tra numeri interi senza segno di ampiezza arbitraria

Per dividere numeri interi con segno di ampiezza arbitraria, si puo' sfruttare il fatto che, i valori assoluti del **quoziente** e del **resto**, sono indipendenti dal segno del **dividendo** e del **divisore**; ad esempio:
```
| 98 / 19 | (Q = 5, R = 3)
```
e:
```
| (-98) / (+19) | (Q = +5, R = +3)
```

Possiamo servirci allora dello stesso procedimento gia' illustrato per l'istruzione **DIV**; a tale proposito, possiamo suddividere l'algoritmo nelle seguenti fasi:

1) memorizzazione del segno dei due operandi;
2) cambiamento di segno (negazione) di eventuali operandi negativi;
3) esecuzione della divisione tra numeri interi senza segno (con **DIV**);
4) adeguamento del risultato in base al segno.

In relazione alla fase n. **4**, osserviamo che i segni del **quoziente** e del **resto** sono dati dalle note regole seguenti:

(+) / (+) (**quoziente** positivo e **resto** positivo);
(+) / (-) (**quoziente** negativo e **resto** positivo);
(-) / (+) (**quoziente** negativo e **resto** negativo);
(-) / (-) (**quoziente** positivo e **resto** negativo).

Se il **quoziente** e il **resto** devono essere positivi, non dobbiamo apportare nessuna modifica ai risultati ottenuti; se, invece, il **quoziente** e/o il **resto** devono essere negativi, dobbiamo eseguire gli opportuni cambiamenti di segno (negazioni).
Dalle considerazioni appena esposte, risulta che, rispetto all'esempio mostrato per **DIV**, abbiamo bisogno anche di un algoritmo per la negazione dei numeri interi con segno di ampiezza arbitraria; quando avremo a disposizione tutti gli strumenti necessari, analizzeremo un esempio pratico.

17.12.5 Effetti provocati da IDIV sugli operandi e sui flags

In base alle considerazioni esposte in precedenza, risulta che l'esecuzione dell'istruzione **IDIV** con operando a **8** bit, modifica i registri **AH** e **AL**; l'esecuzione dell'istruzione **IDIV** con operando a **16** bit, modifica i registri **DX** e **AX**. L'esecuzione dell'istruzione **IDIV** con operando a **32** bit, modifica i registri **EDX** e **EAX**.

La possibilita' di effettuare una divisione tra **Reg** e **Reg**, ci permette di scrivere istruzioni del tipo:

```
idiv ax
```

Come si puo' facilmente constatare, l'esecuzione di questa istruzione produce **DX=0** e **AX=1**; naturalmente, e' importante che prima dell'esecuzione di **DIV**, si esegua l'istruzione **CWD** (inoltre, **AX** deve essere diverso da zero).

L'esecuzione dell'istruzione **IDIV** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; tutti questi **6** campi, assumono un valore indeterminato, e non hanno quindi nessun significato.
Per segnalare eventuali situazioni di errore, la **CPU** utilizza gli stessi metodi gia' illustrati per l'istruzione **DIV**; ovviamente, affinche' non si verifichi un **overflow di divisione**, devono verificarsi le seguenti condizioni:

***** nella divisione tra numeri interi con segno a **8** bit in complemento a **2**, si deve ottenere un **quoziente** compreso tra **-128** e **+127**;
***** nella divisione tra numeri interi con segno a **16** bit in complemento a **2**, si deve ottenere un **quoziente** compreso tra **-32768** e **+32767**;
***** nella divisione tra numeri interi con segno a **32** bit in complemento a **2**, si deve ottenere un **quoziente** compreso tra **-2147483648** e **+2147483647**.

Capitolo 18 - Istruzioni aritmetiche per i numeri BCD

Nel precedente capitolo sono stati illustrati diversi esempi relativi ad istruzioni aritmetiche applicate a numeri interi di ampiezza arbitraria; abbiamo potuto cosi' constatare che, se i numeri su cui si deve operare vengono espressi in binario (cioe', nello stesso codice utilizzato dalla **CPU**), si ottengono procedimenti di calcolo caratterizzati da una notevole semplicita'.
La stessa semplicita' viene riscontrata anche quando operiamo su numeri interi espressi in notazione esadecimale; del resto, sappiamo bene che lo scopo fondamentale della notazione esadecimale, e' quello di permettere la rappresentazione dei numeri binari in un formato molto piu' compatto e molto piu' comodo da maneggiare.
La totale compatibilita' tra notazione binaria e notazione esadecimale, e' una diretta conseguenza del fatto che, come sappiamo, $\mathbf{16=2^4}$. Grazie a questa proprieta' matematica, risulta che una singola cifra esadecimale, corrisponde ad un gruppo di **4** cifre binarie (**1** nibble); viceversa, un gruppo di **4** cifre binarie, corrisponde ad una singola cifra esadecimale.

Il problema che si presenta e' dato dal fatto che, mentre il computer e' stato progettato per lavorare in base **2**, noi esseri umani siamo invece abituati a lavorare in base **10**, ed incontriamo notevoli difficolta' nell'utilizzo di altre basi numeriche; inoltre, sappiamo che il dialogo tra computer ed esseri umani e' reso ancora piu' difficile dal fatto che il numero **10**, non puo' essere espresso come $\mathbf{2^n}$ (potenza di **2** con esponente **n** intero positivo). Se provassimo allora a riscrivere gli algoritmi presentati nel precedente capitolo, in modo da adattarli ai numeri interi in base **10**, andremmo incontro a notevoli complicazioni; si rende necessaria, quindi, la ricerca di una adeguata soluzione a questo importante problema.

In generale, la strada migliore da seguire consiste nel cercare di raggiungere un compromesso, tale da soddisfare le esigenze, sia del computer, sia dell'utente; a tale proposito, nella eventualita' di dover compiere delle elaborazioni su una serie di dati numerici, possiamo seguire un procedimento che puo' essere suddiviso nelle seguenti fasi:

1) input dei dati espressi in base **10**;
2) conversione dei dati in base **2** (o in base **16**);
3) esecuzione delle elaborazioni sui dati;
4) conversione dei risultati in base **10**;
5) output dei risultati espressi in base **10**.

Questo procedimento, soddisfa le esigenze dell'utente in quanto, sia la fase di input, sia la fase di output, si svolgono con l'ausilio della base **10**; allo stesso tempo, vengono soddisfatte anche le esigenze della **CPU** che puo' eseguire le necessarie elaborazioni su dati espressi in base **2**.

18.1 Rappresentazione dei numeri interi decimali in formato BCD

Tra i vari metodi adottati per permettere alla **CPU** di maneggiare facilmente i numeri interi in base **10**, si puo' citare, in particolare, il sistema di rappresentazione in formato **BCD**; la sigla **BCD** sta per **Binary Coded Decimal** (decimale codificato in binario).

Questo sistema di codifica si basa sulla semplice constatazione che un numero esadecimale, formato pero' esclusivamente da cifre comprese tra **0h** e **9h**, ci appare formalmente come se fosse espresso in base **10**; dato, ad esempio, il numero decimale:
```
39674253
```
possiamo pensare di rappresentarlo come:
```
39674253h
```

Attraverso questa tecnica quindi, la **CPU** si serve della base **16** per maneggiare con estrema semplicita' ed efficienza, numeri interi decimali di ampiezza arbitraria che, formalmente, ci appaiono come se fossero espressi in base **10**; in sostanza, la codifica **BCD** permette di estendere alla base **10**, tutti i vantaggi che derivano dall'uso dei numeri esadecimali (vantaggi che abbiamo studiato nei precedenti capitoli).

La codifica **BCD** ha avuto una grande diffusione, soprattutto in campo finanziario, dove si presenta la necessita' di gestire con il computer i bilanci delle aziende, rappresentati spesso da numeri interi di grosse dimensioni; l'utilizzo della base **10** per la gestione col computer di grossi numeri interi (e delle operazioni matematiche ad essi applicate), comporta notevoli complicazioni che possono essere superate, appunto, con l'ausilio della codifica **BCD**.
Anche per altri aspetti, come l'interfacciamento tra **PC** e strumenti di misura, o il salvataggio nella memoria **CMOS** delle informazioni relative alla configurazione hardware del **PC**, si fa' largo uso del sistema di codifica **BCD** (la memoria **CMOS** del **PC**, viene trattata nella sezione **Assembly Avanzato**); possiamo dire quindi che un programmatore **Assembly** che si rispetti, non puo' fare a meno della conoscenza di questo importante argomento.

Tornando all'esempio presentato in precedenza, abbiamo visto che, formalmente, i due numeri **39674253** e **39674253h**, appaiono identici; in realta', sappiamo bene che il numero esadecimale **39674253h**, tradotto in base **10**, corrisponde, non a **39674253**, ma bensi' a:
$(3 * 16^7) + (9 * 16^6) + (6 * 16^5) + (7 * 16^4) + (4 * 16^3) + (2 * 16^2) + (5 * 16^1) + (3 * 16^0)$
cioe':
963068499

Da questa semplice considerazione, si intuiscono subito i problemi che bisogna affrontare con i numeri **BCD**; infatti, appare evidente il fatto che, eseguendo addizioni, sottrazioni, moltiplicazioni e divisioni sui numeri **BCD**, si ottengono risultati che hanno senso in base **16**, ma non in base **10**!
Consideriamo, ad esempio, una somma tra i due numeri **35** e **49**; supponendo che questi due numeri siano espressi in base **10**, otteniamo:
35 + 49 = 84
Supponendo, invece, che questi due numeri siano espressi in formato **BCD** (e quindi, in base **16**), otteniamo:
35h + 49h = 7Eh
E' chiaro quindi che dopo aver eseguito l'addizione tra numeri **BCD**, abbiamo bisogno di un procedimento che ci permetta di trasformare **7Eh** in **84h**, simulando cosi' il risultato che si ottiene in base **10**; lo stesso discorso vale anche per le sottrazioni, moltiplicazioni e divisioni tra numeri **BCD**.

Tutte le **CPU** della famiglia **80x86**, e tutte le **FPU** della famiglia **80x87**, sono dotate di apposite istruzioni esplicitamente rivolte alla gestione dei numeri **BCD**; prima di poter conoscere in dettaglio queste istruzioni, dobbiamo analizzare le convenzioni che vengono adottate per la rappresentazione dei numeri **BCD**.

18.1.1 Rappresentazione dei numeri interi decimali in formato Unpacked BCD

Come e' stato ribadito all'inizio del capitolo, grazie all'eguaglianza **$16=2^4$**, una qualsiasi cifra esadecimale equivale ad un gruppo di **4** bit (nibble); tale equivalenza e' illustrata in Figura 1.

Figura 1 - Equivalenza tra cifre Hex. e nibble

Dec.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Bin.	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

La Figura 1 evidenzia il fatto che se, ad esempio, vogliamo tradurre in binario il numero esadecimale **ADh**, non dobbiamo fare altro che affiancare i due nibble **1010b** e **1101b** (che corrispondono a **Ah** e **Dh**), in modo da ottenere **10101101b**; questa tecnica, non puo' essere invece applicata ai numeri in base **10**, in quanto **10** non e' una potenza di **2** con esponente intero positivo.
Per ovviare a questo problema, possiamo pensare di sfruttare l'equivalenza **Hex. Bin.** illustrata in Figura 1, rappresentando una qualunque cifra decimale compresa tra **0** e **9**, attraverso la corrispondente cifra esadecimale; ogni cifra esadecimale cosi' ottenuta, puo' essere poi memorizzata nel nibble meno significativo di una locazione da **1** byte.

Consideriamo, ad esempio, il numero decimale **38126987**; "simulando" questo numero in base **16**, otteniamo **38126987h**. A questo punto, creiamo una locazione di memoria da **8** byte, e disponiamo le **8** cifre esadecimali di **38126987h**, negli **8** nibble bassi di ciascun byte della locazione stessa.
Nell'ipotesi che la locazione di memoria si trovi all'offset **002Ch** di un segmento dati, otteniamo la situazione illustrata in Figura 2; osserviamo che, nel rispetto della convenzione **little endian**, la cifra meno significativa del numero **BCD** occupa l'indirizzo piu' basso.

Figura 2 - 38126987 "scompattato" in memoria

Hex.	03h	08h	01h	02h	06h	09h	08h	07h
Bin.	00000011b	00001000b	00000001b	00000010b	00000110b	00001001b	00001000b	00000111b
Offset	0033h	0032h	0031h	0030h	002Fh	002Eh	002Dh	002Ch

Come possiamo notare, il nibble piu' significativo di ogni **BYTE** rimane inutilizzato e deve valere **0h** (**0000b**); questo tipo di codifica dei numeri interi in base **10**, viene definito **Unpacked BCD** (**BCD** scompattato).
Nella terminologia utilizzata dalla **Intel**, ogni cifra di un numero **Unpacked BCD** viene definita **ASCII digit** (cifra **ASCII**); il perche' di questa definizione verra' chiarito nel seguito del capitolo.

18.1.2 Rappresentazione dei numeri interi decimali in formato Packed BCD

Come si puo' notare in Figura 2, la codifica **Unpacked BCD** comporta un notevole spreco di memoria; tuttavia, i numeri **Unpacked BCD** presentano il vantaggio di poter essere gestiti in modo molto semplice dalla **CPU**.
Per garantire una rappresentazione piu' compatta dei numeri **BCD**, e' stata definita anche un'altra codifica basata sul fatto che, ogni cifra esadecimale occupa un solo nibble; di conseguenza, in ogni locazione da **1** byte possiamo inserire **2** cifre esadecimali.
In questo modo, il numero **38126987** dell'esempio di Figura 2, viene disposto in memoria come illustrato in Figura 3; si noti che anche in questo caso, viene rispettata la convenzione **little endian**.

Figura 3 - 38126987 "compattato" in memoria

Hex.	38h	12h	69h	87h
Bin.	00111000b	00010010b	01101001b	10000111b
Offset	002Fh	002Eh	002Dh	002Ch

Questo tipo di codifica dei numeri interi in base **10**, viene definito **Packed BCD** (**BCD** compattato); confrontando la Figura 3 con la Figura 2, si nota che un numero in versione **Packed BCD**, occupa la meta' della memoria rispetto alla versione **Unpacked BCD**.
Nella terminologia utilizzata dalla **Intel**, ogni cifra di un numero **Packed BCD** viene definita **decimal digit** (cifra decimale).

18.1.3 Rappresentazione dei numeri interi decimali in formato Packed BCD standard

Per motivi pratici, la **Intel** ha stabilito una serie di convenzioni che definiscono un formato standard per i numeri **Packed BCD**; in base a tali convenzioni, un numero **Packed BCD** standard occupa in memoria una locazione da **10** byte (**1** tenbyte).
Internamente, una locazione di memoria da **10** byte riservata ad un numero **Packed BCD** standard, viene organizzata secondo lo schema di Figura 4; i simboli $\mathbf{C_i}$ (con **i** che va' da **0** a **17**), rappresentano le cifre (ciascuna compresa tra **0h** e **9h**) del numero **Packed BCD**.

Figura 4 - Dato di tipo Packed BCD standard

Contenuto	Segno	C_{17} C_{16}	C_{15} C_{14}	C_{13} C_{12}	C_{11} C_{10}	C_9 C_8	C_7 C_6	C_5 C_4	C_3 C_2	C_1 C_0
Posizione (byte)	9	8	7	6	5	4	3	2	1	0

Come si puo' notare, i **9** byte meno significativi della locazione di memoria, contengono sino a **18** cifre di un numero in formato **Packed BCD**; tali cifre risultano disposte, come al solito, secondo la convenzione **little endian**.
Eventuali cifre non utilizzate, devono valere obbligatoriamente **0h**; ad esempio, se il numero utilizza solo le prime **8** cifre (da $\mathbf{C_0}$ a $\mathbf{C_7}$), tutte le altre cifre (da $\mathbf{C_8}$ a $\mathbf{C_{17}}$) devono valere **0h**.
Il byte piu' significativo della locazione di memoria (decimo byte), codifica il segno del numero **BCD**; il segno positivo e' rappresentato dal codice **00h** (**00000000b**), mentre il segno negativo e' rappresentato dal codice **80h** (**10000000b**).

Possiamo dire quindi che il formato **Packed BCD** standard illustrato in Figura 4, permette di rappresentare numeri interi decimali compresi tra:
`-999999999999999999` e `+999999999999999999`
Risulta, inoltre, che il numero zero puo' essere rappresentato in due modi, e cioe':
`-000000000000000000` e `+000000000000000000`

Tutte le **FPU** della famiglia **80x87**, sono in grado di gestire, direttamente, numeri interi in formato **Packed BCD** standard; a tale proposito, vengono rese disponibili le due istruzioni **FBLD** e **FBSTP**.
L'istruzione **FBLD** (**Fast Packed BCD Load**), legge un numero **Packed BCD** standard dalla **RAM**, e lo carica in un registro della **FPU**; qualsiasi dato caricato in un registro della **FPU**, viene convertito in un formato **floating point** da **10** byte, chiamato **Temporary Real**.
I numeri in formato **Temporary Real**, possono essere utilizzati dalla **FPU** come operandi di espressioni algebriche, logaritmiche, trigonometriche, esponenziali, etc; i risultati che scaturiscono dalle elaborazioni, possono essere poi convertiti dalla **FPU**, in diversi formati.
In particolare, l'istruzione **FBSTP** (**Fast Packed BCD Store and Pop**), legge un numero **Temporary Real** da un registro della **FPU**, e lo memorizza nella **RAM** in formato **Packed BCD** standard.
Tutti i dettagli relativi al funzionamento della **FPU**, e al relativo set di istruzioni, vengono illustrati nella sezione **Assembly Avanzato**.

Da quanto e' stato appena esposto, appare evidente che se disponiamo di una **FPU**, possiamo gestire i numeri **Packed BCD** standard, in modo estremamente semplice e veloce; fortunatamente, tutte le **CPU**, a partire dalla **80486 DX**, sono dotate di **FPU** incorporata.
Anche le **CPU** della famiglia **80x86**, dispongono di apposite istruzioni dedicate espressamente ai numeri **BCD**; tali istruzioni, pero', operano esclusivamente sulle singole cifre di un numero **Packed BCD** o **Unpacked BCD**.
Lo scopo di queste istruzioni, e' quello di "aggiustare" il risultato di addizioni, sottrazioni, moltiplicazioni e divisioni, eseguite su singole cifre di numeri **BCD**; il compito di scrivere tutto il codice necessario per operare su numeri **BCD** di ampiezza arbitraria, viene lasciato quindi al programmatore. In particolare, se vogliamo utilizzare la **CPU** per operare su numeri **Packed BCD** standard, dobbiamo gestire via software tutte le situazioni che si possono presentare in relazione al segno degli operandi; ad esempio, dati i due numeri **+A** e **-B** in formato **Packed BCD** standard, se vogliamo calcolare una somma tra **+A** e **-B**, con **B** maggiore di **A** in valore assoluto, dobbiamo sfruttare il fatto che:

```
(+A) + (-B) = (+A) - (+B) = -((+B) - (+A))
```

Dopo aver calcolato quindi la differenza tra **+B** e **+A**, dobbiamo assegnare il segno negativo (codice **80h**) al risultato.
Tutto cio' e' una diretta conseguenza del fatto che, il formato **Packed BCD** standard, ha il solo scopo di rappresentare simbolicamente numeri interi decimali (positivi e negativi); non avrebbe nessun senso quindi, applicare a questo formato concetti come l'aritmetica modulare, il complemento a **2**, l'estensione del bit di segno, etc. Nel seguito del capitolo, vedremo degli esempi pratici che chiariranno questo aspetto.
Nota importante.
Come e' stato spiegato nel Capitolo 12, se utilizziamo le direttive **TBYTE** o **DT** (**Define Tenbyte**) per definire una variabile intera, e non specifichiamo un suffisso per il valore inizializzante, gli assembler come **MASM** e **TASM** si servono del suffisso predefinito **h**; in sostanza, in assenza di diverse indicazioni da parte del programmatore, l'assembler assume che la variabile rappresenti un numero intero decimale codificato in formato **Packed BCD** standard!
L'assembler provvede, inoltre, a riempire automaticamente con degli zeri a sinistra (**zero extension**), tutti i nibble non utilizzati dal dato appena definito; sfruttando questo aspetto, possiamo definire variabili di tipo **Packed BCD** standard, in questo modo:

```
bcd_number dt 253699842318675
```

Questa definizione crea una locazione di memoria da **10** byte, che contiene il valore esadecimale **00000253699842318675h**.
Per evitare "sorprese", quando si utilizza la direttiva **DT** (o **TBYTE**) si consiglia di specificare sempre il suffisso per il valore inizializzante.

18.2 L'istruzione AAA

Con il mnemonico **AAA** si indica l'istruzione **Ascii adjust AL after Addition** (aggiustamento del contenuto di **AL** dopo una addizione tra due cifre **Unpacked BCD**); lo scopo di questa istruzione e' quello di aggiustare il risultato di una somma tra due cifre codificate in formato **Unpacked BCD**, in modo da farlo apparire formalmente identico a quello che si ottiene in base **10**.
In sostanza, **AAA** presuppone che il programmatore abbia appena eseguito una istruzione **ADD** (o **ADC**) con operandi rappresentati da cifre in formato **Unpacked BCD**; in tal caso, **AAA** legge il risultato della somma, e lo converte in formato **Unpacked BCD**.
L'unica forma lecita per l'istruzione **AAA**, e' la seguente:

```
AAA
```

Come si puo' notare, il programmatore non deve specificare nessun operando esplicito; infatti, la **CPU** assume che la somma si trovi nel registro **AL** (operando **SRC**), e utilizza come operando **DEST**, lo stesso registro **AL**, ed eventualmente anche il registro **AH**.

Per capire il principio di funzionamento di **AAA**, osserviamo innanzi tutto che la somma tra due cifre decimali, e' sempre compresa tra il valore minimo:
```
0 + 0 = 0
```
e il valore massimo:
```
9 + 9 = 18
```
L'istruzione **AAA** deve simulare gli stessi risultati che si ottengono in base **10**, compreso un eventuale riporto; per raggiungere questo scopo, la **CPU** segue un preciso procedimento che viene analizzato attraverso gli esempi seguenti.

a) Il risultato e' compreso tra **0** e **9** (cioe', tra **00h** e **09h**).

Poniamo **AH=0**, **AL=2**, **BL=6**, ed eseguiamo l'istruzione:
```
add al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL + BL = 2 + 6 = 00000010b + 00000110b = 00001000b = 08h = 8, con AF = 0
```
A questo punto, dobbiamo eseguire l'istruzione:
```
aaa
```
L'istruzione **ADD** ha prodotto **AF=0** (nessun riporto dal nibble basso al nibble alto), per cui la **CPU** capisce che il risultato esadecimale e' interamente contenuto nel nibble basso di **AL**.
La **CPU** azzera, in ogni caso, il nibble alto di **AL** e ottiene **AL=08h**; questo valore non supera **09h**, per cui non necessita di nessun aggiustamento (infatti, il risultato esadecimale appare formalmente identico a quello decimale).
La **CPU** lascia inalterato il contenuto di **AH** e segnala l'assenza di riporto decimale ponendo **AF=0**, **CF=0**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=00h:08h**.

b) Il risultato e' compreso tra **10** e **15** (cioe', tra **0Ah** e **0Fh**).

Poniamo **AH=0**, **AL=5**, **BL=9**, ed eseguiamo l'istruzione:
```
add al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL + BL = 5 + 9 = 00000101b + 00001001b = 00001110b = 0Eh = 14, con AF = 0
```
A questo punto, dobbiamo eseguire l'istruzione:
```
aaa
```
L'istruzione **ADD** ha prodotto **AF=0** (nessun riporto dal nibble basso al nibble alto), per cui la **CPU** capisce che il risultato esadecimale e' interamente contenuto nel nibble basso di **AL**.
La **CPU** azzera, in ogni caso, il nibble alto di **AL** e ottiene **AL=0Eh**; questo valore supera **09h**, per

cui necessita di un aggiustamento (infatti, il risultato esadecimale appare formalmente diverso da quello decimale).
La **CPU** somma il valore **06h** ad **AL**, e ottiene:
```
AL = AL + 06h = 0Eh + 06h = 14h
```
La **CPU** azzera di nuovo il nibble alto di **AL**, incrementa di **1** il contenuto di **AH** e segnala il riporto decimale ponendo **AF=1**, **CF=1**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=01h:04h**.

c) Il risultato e' compreso tra **16** e **18** (cioe', tra **10h** e **12h**).

Poniamo **AH=0**, **AL=9**, **BL=9**, ed eseguiamo l'istruzione:
```
add al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL + BL = 9 + 9 = 00001001b + 00001001b = 00010010b = 12h = 18, con AF = 1
```
A questo punto, dobbiamo eseguire l'istruzione:
```
aaa
```
L'istruzione **ADD** ha prodotto **AF=1** (riporto dal nibble basso al nibble alto), per cui la **CPU** capisce che il risultato esadecimale occupa entrambi i nibble di **AL**; in questo caso, il risultato e' sicuramente superiore a **09h**, per cui necessita di un aggiustamento (infatti, il risultato esadecimale appare formalmente diverso da quello decimale).
La **CPU** somma il valore **06h** ad **AL**, e ottiene:
```
AL = AL + 06h = 12h + 06h = 18h
```
La **CPU** azzera il nibble alto di **AL**, incrementa di **1** il contenuto di **AH** e segnala il riporto decimale ponendo **AF=1**, **CF=1**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=01h:08h**.

Riassumendo, se **AF=0** e **CF=0**, allora la somma in formato **Unpacked BCD** e' costituita unicamente dalla cifra contenuta nel nibble basso di **AL** (nessun riporto decimale); se, invece, **AF=1** e **CF=1**, allora la somma in formato **Unpacked BCD** e' costituita dalla cifra contenuta nel nibble basso di **AL**, e da un riporto decimale di **1**.

Non e' obbligatorio l'azzeramento di **AH** prima di una istruzione **AAA**; il programmatore e' libero di gestire come meglio crede, il contenuto di tale registro.

Ci si puo' chiedere che legame ci sia tra il codice ASCII e i numeri in formato **Unpacked BCD**; tale legame e' dovuto ad un particolare effetto collaterale prodotto dall'istruzione **AAA**.
Per chiarire questo aspetto, proviamo a calcolare la somma, in formato **Unpacked BCD**, tra le due cifre **3** e **5**; al loro posto, utilizziamo pero' i codici ASCII dei corrispondenti simboli **'3'** e **'5'**, e cioe', **33h** e **35h**.
Come si puo' notare, i nibble bassi dei due codici **33h** e **35h**, valgono, rispettivamente, **3h** e **5h**, proprio come le due cifre da sommare; eseguendo ora l'istruzione **ADD** con i due operandi **'3'** e **'5'** (o, direttamente, **33h** e **35h**), otteniamo:
```
AL = 33h + 35h = 68h, con AF = 0
```
In presenza ora di una istruzione **AAA**, e in base a quanto abbiamo visto in precedenza, la **CPU** azzera il nibble alto di **AL**, lascia inalterato il contenuto di **AH** e segnala l'assenza di riporto decimale ponendo **AF=0**, **CF=0**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=00h:08h** (assumendo **AH=0** prima dell'esecuzione di **AAA**).
Tale risultato e' identico a quello che avremmo ottenuto sommando **3** e **5**; in sostanza, l'istruzione **AAA** opera, indifferentemente, su cifre **Unpacked BCD** o sui codici ASCII dei simboli corrispondenti alle cifre stesse!

18.2.1 Somma tra numeri Unpacked BCD di ampiezza arbitraria

Sulla base di quanto e' stato esposto in precedenza, possiamo intuire che la somma tra numeri **Unpacked BCD** di ampiezza arbitraria, puo' essere svolta in modo semplicissimo; a tale proposito, vediamo subito un esempio pratico.

Supponiamo di voler eseguire la seguente somma tra numeri interi decimali da **16** cifre ciascuno:

```
5286548936543797 +
2943567238661142 =
------------------
8230116175204939
```

Convertendo tutto in formato **Unpacked BCD**, dobbiamo ottenere:

```
05020806050408090306050403070907h +
02090403050607020308060601010402h =
-----------------------------------
08020300010106010705020004090309h
```

Eseguendo la prima somma **Unpacked BCD** con **ADD**, si ha:
`AL = 07h + 02h = 09h, con AF = 0`
Una successiva istruzione **AAA** produce **AL=09h**, **AF=0** e **CF=0**.

Eseguendo la seconda somma **Unpacked BCD** con **ADC**, si ha:
`AL = 09h + 04h + CF = 0Dh + 0h = 0Dh, con AF = 0`
Una successiva istruzione **AAA** produce **AL=03h**, **AF=1** e **CF=1**.

Eseguendo la terza somma **Unpacked BCD** con **ADC**, si ha:
`AL = 07h + 01h + CF = 08h + 1h = 09h, con AF = 0`
Una successiva istruzione **AAA** produce **AL=09h**, **AF=0** e **CF=0**.

A questo punto e' inutile continuare in quanto abbiamo capito che il procedimento da seguire, e' del tutto simile a quello illustrato nel precedente capitolo per l'istruzione **ADC**; l'unica differenza sta nel fatto che, dopo ogni istruzione **ADD** (o **ADC**) con operando **AL**, dobbiamo aggiungere una istruzione **AAA**.
Terminata l'ultima somma (tra le cifre piu' significative), dobbiamo consultare **CF**; se **CF=0**, il risultato e' valido, mentre se **CF=1**, il risultato eccede le **16** cifre **Unpacked BCD**.

Per definire i vari dati da elaborare, possiamo servirci della direttiva **DB** (**Define Byte**) nel seguente modo:

```
ubcdNum1    db    7, 9, 7, 3, 4, 5, 6, 3, 9, 8, 4, 5, 6, 8, 2, 5
ubcdNum2    db    2, 4, 1, 1, 6, 6, 8, 3, 2, 7, 6, 5, 3, 4, 9, 2
ubcdSomma   db    16 dup (0)
```

E' importante ricordare che, in questo tipo di definizioni, la cifra meno significativa e' quella piu' a sinistra.
Il codice che esegue la somma assume il seguente aspetto:

```
; somma delle prime cifre con ADD e AAA

mov      al, ubcdNum1[0]          ; al = ubcdNum1[0]
add      al, ubcdNum2[0]          ; al = ubcdNum1[0] + ubcdNum2[0]
aaa                               ; aggiustamento ASCII dell'addizione
mov      ubcdSomma[0], al         ; ubcdSomma[0] = al

; somma delle seconde cifre con ADC e AAA

mov      al, ubcdNum1[1]          ; al = ubcdNum1[1]
adc      al, ubcdNum2[1]          ; al = ubcdNum1[1] + ubcdNum2[1] + CF
aaa                               ; aggiustamento ASCII dell'addizione
mov      ubcdSomma[1], al         ; ubcdSomma[1] = al

; somma delle terze cifre con ADC e AAA

mov      al, ubcdNum1[2]          ; al = ubcdNum1[2]
adc      al, ubcdNum2[2]          ; al = ubcdNum1[2] + ubcdNum2[2] + CF
aaa                               ; aggiustamento ASCII dell'addizione
mov      ubcdSomma[2], al         ; ubcdSomma[2] = al

; ... e cosi' via per le altre 13 cifre

call     showCPU                  ; verifica dei flags
```

Dopo l'esecuzione dell'ultima somma (tra le due cifre piu' significative), consultiamo **CF** per sapere se c'e' stato un riporto finale; nel nostro caso, **CF=0**, per cui il risultato e' valido cosi' com'e'.

Se vogliamo visualizzare il risultato in formato **Unpacked BCD**, possiamo servirci di **writeHex8**; partendo dalla cifra piu' significativa di **ubcdSomma**, possiamo scrivere:

```
mov      al, ubcdSomma[15]        ; al = ubcdSomma[15]
mov      dx, 0400h                ; riga 4, colonna 0
call     writeHex8                ; visualizza al

mov      al, ubcdSomma[14]        ; al = ubcdSomma[14]
mov      dx, 0402h                ; riga 4, colonna 2
call     writeHex8                ; visualizza al

mov      al, ubcdSomma[13]        ; al = ubcdSomma[13]
mov      dx, 0404h                ; riga 4, colonna 4
call     writeHex8                ; visualizza al

; ... e cosi' via
```

18.2.2 Effetti provocati da AAA sugli operandi e sui flags

Se la somma tra due cifre in formato **Unpacked BCD** e' compresa tra **0** e **9**, l'esecuzione dell'istruzione **AAA** provoca la modifica del solo registro **AL**; se, invece, la somma tra due cifre in formato **Unpacked BCD** e' compresa tra **10** e **18**, l'esecuzione dell'istruzione **AAA** provoca la modifica dei registri **AL** e **AH**.

L'esecuzione dell'istruzione **AAA** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; i campi **PF**, **ZF**, **SF** e **OF**, assumono un valore indeterminato, e non hanno quindi nessun significato.
La **CPU** pone **AF=0** e **CF=0**, per segnalare che la somma tra due cifre in formato **Unpacked BCD** e' compresa tra **0** e **9**; la **CPU** pone, invece, **AF=1** e **CF=1**, per segnalare che la somma tra due cifre in formato **Unpacked BCD** e' compresa tra **10** e **18** (riporto decimale).

18.3 L'istruzione <u>AAS</u>

Con il mnemonico **AAS** si indica l'istruzione **Ascii adjust AL after Subtraction** (aggiustamento del contenuto di **AL** dopo una sottrazione tra due cifre **Unpacked BCD**); lo scopo di questa istruzione e' quello di aggiustare il risultato di una differenza tra due cifre codificate in formato **Unpacked BCD**, in modo da farlo apparire formalmente identico a quello che si ottiene in base **10**.
In sostanza, **AAS** presuppone che il programmatore abbia appena eseguito una istruzione **SUB** (o **SBB**) con operandi rappresentati da cifre in formato **Unpacked BCD**; in tal caso, **AAS** legge il risultato della sottrazione, e lo converte in formato **Unpacked BCD**.
L'unica forma lecita per l'istruzione **AAS**, e' la seguente:

```
AAS
```

Come si puo' notare, il programmatore non deve specificare nessun operando esplicito; infatti, la **CPU** assume che la differenza si trovi nel registro **AL** (operando **SRC**), e utilizza come operando **DEST**, lo stesso registro **AL**, ed eventualmente anche il registro **AH**.

Per capire il principio di funzionamento di **AAS**, osserviamo innanzi tutto che, nel calcolo della differenza tra due cifre decimali, se il **minuendo** e' maggiore o uguale al **sottraendo**, si ottiene sempre un risultato compreso tra **0** e **9**, senza prestito dalla cifra successiva del **minuendo**; se, invece, il **minuendo** e' minore del **sottraendo**, si ottiene sempre un risultato compreso tra **1** e **9**, con prestito di **1** dalla cifra successiva del **minuendo**.
L'istruzione **AAS** deve simulare gli stessi risultati che si ottengono in base **10**, compreso un eventuale prestito; per raggiungere questo scopo, la **CPU** segue un preciso procedimento che viene analizzato attraverso gli esempi seguenti.

a) Il **minuendo** e' maggiore o uguale al **sottraendo**.

Poniamo **AH=0**, **AL=8**, **BL=3**, ed eseguiamo l'istruzione:
```
sub al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL - BL = 8 - 3 = 00001000b - 00000011b = 00000101b = 05h = 5, con AF = 0
```
A questo punto, dobbiamo eseguire l'istruzione:
```
aas
```
L'istruzione **SUB** ha prodotto **AF=0** (nessun prestito dal nibble alto al nibble basso), per cui la **CPU** capisce che il risultato non necessita di nessun aggiustamento (infatti, il risultato esadecimale appare formalmente identico a quello decimale).
La **CPU** azzera, in ogni caso, il nibble alto di **AL** e ottiene **AL=05h**.

La **CPU** lascia inalterato il contenuto di **AH** e segnala l'assenza di prestito decimale ponendo **AF=0**, **CF=0**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=00h:05h**.

b) Il **minuendo** e' minore del **sottraendo**.

Poniamo **AH=0**, **AL=5**, **BL=9**, ed eseguiamo l'istruzione:
```
sub al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL - BL = 5 - 9 = 00000101b - 00001001b = 11111100b = FCh, con AF = 1
```
In base **10**, la sottrazione **5-9** produce il risultato **6**, con prestito di **1**.
A questo punto, dobbiamo eseguire l'istruzione:
```
aas
```
L'istruzione **SUB** ha prodotto **AF=1** (prestito dal nibble alto al nibble basso), per cui la **CPU** capisce che il risultato necessita di un aggiustamento (infatti, il risultato esadecimale appare formalmente diverso da quello decimale).
La **CPU** sottrae il valore **06h** ad **AL**, e ottiene:
```
AL = AL - 06h = FCh - 06h = F6h
```
La **CPU** azzera il nibble alto di **AL**, decrementa di **1** il contenuto di **AH** e segnala il prestito decimale ponendo **AF=1**, **CF=1**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=FFh:06h**.

Riassumendo, se **AF=0** e **CF=0**, allora la differenza in formato **Unpacked BCD** e' costituita unicamente dalla cifra contenuta nel nibble basso di **AL** (nessun prestito decimale); se, invece, **AF=1** e **CF=1**, allora la differenza in formato **Unpacked BCD** e' costituita dalla cifra contenuta nel nibble basso di **AL**, e da un prestito decimale di **1**.

Non e' obbligatorio l'azzeramento di **AH** prima di una istruzione **AAS**; il programmatore e' libero di gestire come meglio crede, il contenuto di tale registro.

Anche per **AAS**, vale l'effetto collaterale gia' descritto per **AAA**; in sostanza, l'istruzione **AAS** opera, indifferentemente, su cifre **Unpacked BCD** o sui codici ASCII dei simboli corrispondenti alle cifre stesse!

18.3.1 Differenza tra numeri Unpacked BCD di ampiezza arbitraria

Sulla base di quanto e' stato esposto in precedenza, possiamo intuire che la differenza tra numeri **Unpacked BCD** di ampiezza arbitraria, puo' essere svolta in modo semplicissimo; a tale proposito, vediamo subito un esempio pratico.

Supponiamo di voler eseguire la seguente differenza tra numeri interi decimali da **20** cifre ciascuno:

```
86436578912345734942 -
35873863219546738871 =
----------------------
50562715692798996071
```

Convertendo tutto in formato **Unpacked BCD**, dobbiamo ottenere:

```
0806040306050708090102030405070304090402h -
0305080703080603020109050406070308080701h =
------------------------------------------
0500050602070105060902070908090906000701h
```

Eseguendo la prima differenza **Unpacked BCD** con **SUB**, si ha:
`AL = 02h - 01h = 01h, con AF = 0`
Una successiva istruzione **AAS** produce **AL=01h**, **AF=0** e **CF=0**.

Eseguendo la seconda differenza **Unpacked BCD** con **SBB**, si ha:
`AL = 04h - 07h - CF = FDh - 0h = FDh, con AF = 1`
Una successiva istruzione **AAS** produce **AL=07h**, **AF=1** e **CF=1**.

Eseguendo la terza differenza **Unpacked BCD** con **SBB**, si ha:
`AL = 09h - 08h - CF = 01h - 1h = 00h, con AF = 0`
Una successiva istruzione **AAS** produce **AL=00h**, **AF=0** e **CF=0**.

A questo punto e' inutile continuare in quanto abbiamo capito che il procedimento da seguire, e' del tutto simile a quello illustrato nel precedente capitolo per l'istruzione **SBB**; l'unica differenza sta nel fatto che, dopo ogni istruzione **SUB** (o **SBB**) con operando **AL**, dobbiamo aggiungere una istruzione **AAS**.
Terminata l'ultima differenza (tra le cifre piu' significative), dobbiamo consultare **CF**; se **CF=0**, il risultato e' valido, mentre se **CF=1**, si e' verificato un prestito finale (**minuendo** minore del **sottraendo**).

Come e' stato gia' detto, la codifica **BCD** ha il solo scopo di permettere una rappresentazione simbolica dei numeri interi decimali; di conseguenza, il programmatore ha il compito di gestire le varie situazioni che si possono presentare in relazione al segno degli operandi.
Ad esempio, prima di eseguire una sottrazione il cui **minuendo** e' minore del **sottraendo**, dobbiamo scambiare di posto i due operandi, in modo da ottenere un risultato positivo; a questo risultato, dobbiamo poi assegnare il segno negativo. Naturalmente, e' nostro compito definire anche il metodo per la codifica del segno; prendendo spunto, ad esempio, dai numeri **Packed BCD** standard, possiamo utilizzare la cifra piu' significativa di un numero **Unpacked BCD** (di ampiezza definita), per memorizzare il segno del numero stesso.

Se vogliamo implementare in pratica l'esempio precedente, possiamo procedere con la definizione dei dati nel seguente modo:

```
ubcdNum1    db    2, 4, 9, 4, 3, 7, 5, 4, 3, 2, 1, 9, 8, 7, 5, 6, 3, 4, 6, 8
ubcdNum2    db    1, 7, 8, 8, 3, 7, 6, 4, 5, 9, 1, 2, 3, 6, 8, 3, 7, 8, 5, 3
ubcdDiff    db    20 dup (0)
```

Il codice che esegue la differenza assume il seguente aspetto:

```
; differenza tra le prime cifre con SUB e AAS

mov      al, ubcdNum1[0]         ; al = ubcdNum1[0]
sub      al, ubcdNum2[0]         ; al = ubcdNum1[0] - ubcdNum2[0]
aas                              ; aggiustamento ASCII della sottrazione
mov      ubcdDiff[0], al         ; ubcdDiff[0] = al

; differenza tra le seconde cifre con SBB e AAS

mov      al, ubcdNum1[1]         ; al = ubcdNum1[1]
sbb      al, ubcdNum2[1]         ; al = ubcdNum1[1] - ubcdNum2[1] - CF
aas                              ; aggiustamento ASCII della sottrazione
mov      ubcdDiff[1], al         ; ubcdDiff[1] = al

; differenza tra le terze cifre con SBB e AAS

mov      al, ubcdNum1[2]         ; al = ubcdNum1[2]
sbb      al, ubcdNum2[2]         ; al = ubcdNum1[2] - ubcdNum2[2] - CF
aas                              ; aggiustamento ASCII della sottrazione
mov      ubcdDiff[2], al         ; ubcdDiff[2] = al

; ... e cosi' via per le altre 17 cifre

call     showCPU                 ; verifica dei flags
```

Dopo l'esecuzione dell'ultima differenza (tra le due cifre piu' significative), consultiamo **CF** per sapere se c'e' stato un prestito finale; nel nostro caso, **CF=0**, per cui il risultato e' valido cosi' com'e'.

Se vogliamo visualizzare il risultato in formato **Unpacked BCD**, possiamo procedere come nell'esempio su **AAA**; in questo caso, l'output inizia da **ubcdDiff[19]**.

18.3.2 Effetti provocati da AAS sugli operandi e sui flags

Se la differenza tra due cifre in formato **Unpacked BCD** non richiede nessun prestito, l'esecuzione dell'istruzione **AAS** provoca la modifica del solo registro **AL**; se, invece, la differenza tra due cifre in formato **Unpacked BCD** richiede un prestito, l'esecuzione dell'istruzione **AAS** provoca la modifica dei registri **AL** e **AH**.

L'esecuzione dell'istruzione **AAS** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; i campi **PF**, **ZF**, **SF** e **OF**, assumono un valore indeterminato, e non hanno quindi nessun significato.
La **CPU** pone **AF=0** e **CF=0**, per segnalare che la differenza tra due cifre in formato **Unpacked BCD** non ha richiesto nessun prestito; la **CPU** pone, invece, **AF=1** e **CF=1**, per segnalare che la differenza tra due cifre in formato **Unpacked BCD** ha richiesto un prestito.

18.4 L'istruzione AAM

Con il mnemonico **AAM** si indica l'istruzione **Ascii adjust AX after Multiplication** (aggiustamento del contenuto di **AX** dopo una moltiplicazione tra due cifre **Unpacked BCD**); lo scopo di questa istruzione e' quello di aggiustare il risultato di un prodotto tra due cifre codificate in formato **Unpacked BCD**, in modo da farlo apparire formalmente identico a quello che si ottiene in base **10**.
In sostanza, **AAM** presuppone che il programmatore abbia appena eseguito una istruzione **MUL** con operandi rappresentati da cifre in formato **Unpacked BCD**; in tal caso, **AAM** legge il risultato della moltiplicazione, e lo converte in formato **Unpacked BCD**.
In virtu' del fatto che i numeri **BCD** non seguono le leggi dell'aritmetica modulare, e' necessario sottolineare che **AAM** produce un risultato valido solo se viene eseguita subito dopo una istruzione **MUL** (prodotto tra numeri interi senza segno); se, invece, si esegue **AAM** dopo una istruzione **IMUL**, si ottengono risultati privi di senso!
L'unica forma lecita per l'istruzione **AAM**, e' la seguente:

```
AAM
```

Come si puo' notare, il programmatore non deve specificare nessun operando esplicito; infatti, la **CPU** assume che il prodotto si trovi nel registro **AX** (operando **SRC**), e utilizza come operando **DEST**, lo stesso registro **AX**.

Per capire il principio di funzionamento di **AAM**, osserviamo innanzi tutto che il prodotto tra due cifre decimali, e' un numero formato, al massimo, da due cifre. Tale prodotto, e' sempre compreso tra il valore minimo:

```
0 * 0 = 0
```

e il valore massimo:

```
9 * 9 = 81
```

Per convertire il risultato decimale in formato **Unpacked BCD**, l'istruzione **AAM** si serve del **metodo delle divisioni successive**, gia' illustrato nel Capitolo 4; tale metodo prevede che, dividendo ripetutamente un numero intero in base **b1**, per una qualsiasi base **b2** (divisione intera), si ottengono una serie di resti che rappresentano la sequenza delle cifre in base **b2** del numero stesso. Le divisioni terminano quando si ottiene un quoziente nullo; l'ultimo resto ottenuto rappresenta la cifra piu' significativa del numero appena convertito in base **b2**.
Applichiamo ora questo metodo ad un numero intero decimale a due sole cifre, da dividere per la sua stessa base **b=10**; a tale proposito, consideriamo il seguente prodotto decimale:

```
6 * 9 = 54
```

Dividendo ora **54** per la sua base **10** (divisione intera), si ottiene:

```
54 / 10, Q = 5, R = 4
```

Come possiamo notare, il **quoziente** e il **resto** rappresentano, rispettivamente, la cifra piu' significativa e quella meno significativa di **54**!
Una volta che il numero e' stato scomposto nelle sue due cifre, e' facile codificarlo in formato **Unpacked BCD**; vediamo subito alcuni esempi pratici.

Poniamo **AL=3**, **BL=2**, ed eseguiamo l'istruzione:

```
mul bl
```

In presenza di questa istruzione, la **CPU** calcola:

```
AH:AL = AL * BL = 3 * 2 = 6
```

A questo punto, dobbiamo eseguire l'istruzione:

```
aam
```

La **CPU** divide il prodotto **6** per la sua base **10**, e ottiene:

```
Temp8:Temp8 = AH:AL / 10 = 6 / 10, Q = 0, R = 6
```

Successivamente, la **CPU** pone **AL=R=06h** e **AH=Q=00h**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=00h:06h**.

Poniamo **AL=9**, **BL=6**, ed eseguiamo l'istruzione:
```
mul bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AH:AL = AL * BL = 9 * 6 = 54
```
A questo punto, dobbiamo eseguire l'istruzione:
```
aam
```
La **CPU** divide il prodotto **54** per la sua base **10**, e ottiene:
```
Temp8:Temp8 = AH:AL / 10 = 54 / 10, Q = 5, R = 4
```
Successivamente, la **CPU** pone **AL=R=04h** e **AH=Q=05h**; alla fine otteniamo il risultato **Unpacked BCD** dato da **AH:AL=05h:04h**.

A differenza di quanto accade per **AAA** e **AAS**, l'istruzione **AAM** opera, esclusivamente, su cifre in formato **Unpacked BCD**; non e' possibile quindi utilizzare i codici ASCII dei simboli corrispondenti alle cifre stesse!

18.4.1 Codifica binaria di numeri interi non decimali

L'istruzione **AAM** si serve della base predefinita **10**; esiste pero' la possibilita' di codificare in binario, numeri espressi in qualsiasi altra base. A tale proposito, e' necessario servirsi, obbligatoriamente, di un apposito codice macchina rappresentato dalla sequenza:
```
11010100b Imm8 = D4h Imm8
```
Il valore immediato **Imm8** rappresenta la base da utilizzare (ad esempio, **8**, **8h** o **10q** per la notazione ottale); ovviamente, l'utilizzo dell'**Opcode D4h** con base **0**, provoca un **overflow di divisione**!
Supponiamo, ad esempio, di voler codificare in binario scompattato, numeri espressi in base **8**; in tal caso, potremmo parlare di formato **Unpacked BCO** (**Binary Coded Octal**). In particolare, consideriamo il seguente prodotto:
```
3 * 7 = 21 = 25q
```
(e' ovvio che, in notazione ottale, il prodotto deve svolgersi tra cifre comprese tra **0** e **7**).
Se ora vogliamo convertire il risultato in formato **Unpacked BCO**, possiamo scrivere le seguenti istruzioni:

```
mov      al, 3        ; al = 3
mov      bl, 7        ; bl = 7
mul      bl           ; ah:al = al * bl
db       0D4h, 10q    ; AAM con base ottale
```

Provando ora a visualizzare il contenuto dei due registri **AH** e **AL**, otteniamo proprio **AH:AL=02h:05h**!

18.4.2 Prodotto tra numeri Unpacked BCD di ampiezza arbitraria

Sulla base di quanto e' stato esposto in precedenza, possiamo intuire che il prodotto tra numeri **Unpacked BCD** di ampiezza arbitraria, puo' essere svolto in modo semplicissimo; a tale proposito, vediamo subito un esempio pratico.

Supponiamo di voler eseguire il seguente prodotto:
`6973 * 8 = 55784`
Moltiplicando un numero a **4** cifre per un numero a **1** cifra, otteniamo un risultato che richiede, al massimo **4+1=5** cifre.
Convertendo tutto in formato **Unpacked BCD**, dobbiamo ottenere:
`06090703h * 08h = 0505070804h`
Moltiplicando un numero da **4 BYTE** per un numero da **1 BYTE**, otteniamo un risultato che richiede, al massimo **4+1=5 BYTE**.

Eseguendo il primo prodotto **Unpacked BCD** con **MUL**, si ha:
`AH:AL = 03h * 08h = 18h = 24`
Una successiva istruzione **AAM** produce **AH:AL=02h:04h**, cioe', **04h** con riporto di **02h**.

Eseguendo il secondo prodotto **Unpacked BCD** con **MUL**, si ha:
`AH:AL = 07h * 08h = 38h = 56`
Una successiva istruzione **AAM** produce **AH:AL=05h:06h**, cioe', **06h** con riporto di **05h**.
Alla cifra **06h** dobbiamo sommare il precedente riporto **02h**; a tale proposito, servendoci dell'istruzione **ADD**, otteniamo:
`AL = 06h + 02h = 08h, con AF = 0`
Una successiva istruzione **AAA** produce **AL=08h**, con **AF=0** e **CF=0**.
La precedente somma non provoca un **carry**, per cui il riporto **05h** (incrementato con **ADC**) rimane invariato.

Eseguendo il terzo prodotto **Unpacked BCD** con **MUL**, si ha:
`AH:AL = 09h * 08h = 48h = 72`
Una successiva istruzione **AAM** produce **AH:AL=07h:02h**, cioe', **02h** con riporto di **07h**.
Alla cifra **02h** dobbiamo sommare il precedente riporto **05h**; a tale proposito, servendoci dell'istruzione **ADD**, otteniamo:
`AL = 02h + 05h = 07h, con AF = 0`
Una successiva istruzione **AAA** produce **AL=07h**, con **AF=0** e **CF=0**.
La precedente somma non provoca un **carry**, per cui il riporto **07h** (incrementato con **ADC**) rimane invariato.

Eseguendo il quarto prodotto **Unpacked BCD** con **MUL**, si ha:
`AH:AL = 06h * 08h = 30h = 48`
Una successiva istruzione **AAM** produce **AH:AL=04h:08h**, cioe', **08h** con riporto di **04h**.
Alla cifra **08h** dobbiamo sommare il precedente riporto **07h**; a tale proposito, servendoci dell'istruzione **ADD**, otteniamo:
`AL = 08h + 07h = 0Fh, con AF = 0`
Una successiva istruzione **AAA** produce **AL=05h**, con **AF=1** e **CF=1**.

L'ultimo riporto **04h** (incrementato con **ADC**) diventa **05h**, e rappresenta la cifra piu' significativa del risultato.

Unendo le **5** cifre appena calcolate, otteniamo proprio **0505070804h**!.

Come possiamo notare, il procedimento da seguire e' del tutto simile a quello utilizzato nel precedente capitolo per l'istruzione **MUL**; il codice e' molto piu' lungo a causa del fatto che dobbiamo operare sulle singole cifre **Unpacked BCD**.

Il procedimento appena illustrato, puo' essere notevolmente semplificato eliminando tutte le istruzioni **AAA** e **ADC**, e posizionando **AAM** dopo **ADD**; infatti, osserviamo innanzi tutto che **AAM** opera correttamente su un qualsiasi prodotto compreso tra **0** e **99**. Il prodotto massimo che possiamo ottenere tra due cifre decimali e':
`9 * 9 = 81` (cioe', **1** con riporto di **8**).
Se il precedente riporto era **9**, cioe' il massimo possibile, l'istruzione **ADD** produce:
`81 + 9 = 90` (cioe', **0** con riporto di **9**).
Tale risultato e' chiaramente inferiore a **99**; a questo punto, una istruzione **AAM** produce:
`AH:AL = 09h:00h` (cioe', **0** con riporto di **9**).

Se vogliamo implementare in pratica l'esempio precedente, possiamo procedere con la definizione dei dati nel seguente modo:

```
ubcdFatt1   db    3, 7, 9, 6
ubcdFatt2   db    8
ubcdProd    db    5 dup (0)
```

In questo caso particolare, il dato **ubcdFatt1** puo' essere definito anche come:
`ubcdFatt1 dd 06090703h`
In tal caso, per maneggiare **ubcdFatt1** dobbiamo servirci dell'operatore **BYTE PTR**.

Il codice che esegue il prodotto assume il seguente aspetto:

```
; prodotto tra le prime cifre con MUL e AAM

mov      al, ubcdFatt2           ; al = ubcdFatt2
mul      ubcdFatt1[0]            ; ah:al = ubcdFatt2 * ubcdFatt1[0]
aam                              ; aggiustamento ASCII del prodotto
mov      cl, ah                  ; salva il riporto attuale
mov      ubcdProd[0], al         ; ubcdProd[0] = al

; prodotto tra le seconde cifre con MUL, ADD e AAM

mov      al, ubcdFatt2           ; al = ubcdFatt2
mul      ubcdFatt1[1]            ; ah:al = ubcdFatt2 * ubcdFatt1[1]
add      al, cl                  ; somma al con il riporto precedente
aam                              ; aggiustamento ASCII del prodotto
mov      cl, ah                  ; salva il riporto attuale
mov      ubcdProd[1], al         ; ubcdProd[1] = al

; prodotto tra le terze cifre con MUL, ADD e AAM

mov      al, ubcdFatt2           ; al = ubcdFatt2
mul      ubcdFatt1[2]            ; ah:al = ubcdFatt2 * ubcdFatt1[2]
add      al, cl                  ; somma al con il riporto precedente
aam                              ; aggiustamento ASCII del prodotto
mov      cl, ah                  ; salva il riporto attuale
mov      ubcdProd[2], al         ; ubcdProd[2] = al

; prodotto tra le quarte cifre con MUL, ADD e AAM

mov      al, ubcdFatt2           ; al = ubcdFatt2
```

```
mul      ubcdFatt1[3]           ; ah:al = ubcdFatt2 * ubcdFatt1[3]
add      al, cl                 ; somma al con il riporto precedente
aam                             ; aggiustamento ASCII del prodotto
mov      ubcdProd[3], al        ; ubcdProd[3] = al
mov      ubcdProd[4], ah        ; ubcdProd[4] = ah (ultimo riporto)
```

Se vogliamo visualizzare il risultato in formato **Unpacked BCD**, possiamo procedere come nell'esempio sull'istruzione **AAA**; in questo caso, l'output inizia da **ubcdProd[4]**.

Cosa succede se anche il **moltiplicatore** e' formato da due o piu' cifre decimali?
Anche in un caso del genere dobbiamo ricorrere allo stesso algoritmo che si segue con carta e penna; il codice da scrivere diventa notevolmente piu' lungo, principalmente a causa del fatto che, i vari prodotti parziali, devono essere sommati tra loro sempre in formato **Unpacked BCD**.
Consideriamo, ad esempio, il seguente prodotto:

```
36742861 * 568 =
----------------
   293942888
  220457166
 183714305
----------------
 20869945048
```

Moltiplicando un numero a **8** cifre per un numero a **3** cifre, otteniamo un prodotto che richiede, al massimo, **8+3=11** cifre; traducendo tutto in formato **Unpacked BCD**, dobbiamo ottenere:

```
0306070402080601h * 050608h =
-----------------------------
   0000020903090402080808h
   0002020004050701060600h
   0108030701040300050000h
-----------------------------
   0200080609090405000408h
```

Moltiplicando un numero da **8 BYTE** per un numero da **3 BYTE**, otteniamo un prodotto che richiede, al massimo, **8+3=11 BYTE**.
Applicando ora il metodo illustrato nel precedente esempio, otteniamo tre prodotti parziali (uno per ogni cifra del **moltiplicatore**); la somma, in formato **Unpacked BCD**, tra i primi due prodotti parziali, produce:

```
0000020903090402080808h +
0002020004050701060600h =
-------------------------
0002040908050104050408h
```

La somma, in formato **Unpacked BCD**, tra il risultato appena ottenuto e il terzo prodotto parziale, produce:

```
0002040908050104050408h +
0108030701040300050000h =
-------------------------
0200080609090405000408h
```

Analizzando la struttura delle somme appena svolte, si puo' notare che per far scorrere verso sinistra i **BYTE** dei vari prodotti parziali, non abbiamo la necessita' di ricorrere alle istruzioni di shifting; tutto cio' e' una diretta conseguenza del fatto che i numeri **Unpacked BCD**, sono suddivisi appunto in **BYTE**, e quindi sono facilmente maneggiabili anche senza l'ausilio delle istruzioni logiche (illustrate nel capitolo successivo).

Un'ultima considerazione riguarda l'eventualita' di dover moltiplicare tra loro, numeri **Unpacked BCD** con segno.
Come e' stato gia' sottolineato, tutte le operazioni in formato **Unpacked BCD** devono svolgersi tra numeri senza segno; alla fine, si deve procedere alla opportuna codifica del segno dei risultati che scaturiscono dalle operazioni stesse.

18.4.3 Effetti provocati da AAM sugli operandi e sui flags

L'esecuzione dell'istruzione **AAM** provoca la modifica del solo registro **AX**.

L'esecuzione dell'istruzione **AAM** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; i campi **CF**, **AF**, **OF**, assumono un valore indeterminato, e non hanno quindi nessun significato logico.
I campi **PF**, **ZF** e **SF** forniscono, invece, informazioni ordinarie relative al numero binario memorizzato in **AL** da **AAM**; in sostanza, **PF** indica se **AL** contiene un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il contenuto di **AL** vale zero, **SF** indica se il bit piu' significativo di **AL** vale **1**. Generalmente, il contenuto di questi tre flags viene ignorato.

18.5 L'istruzione AAD

Con il mnemonico **AAD** si indica l'istruzione **Ascii adjust AX before Division** (aggiustamento del contenuto di **AX** prima di una divisione); lo scopo di questa istruzione e' quello di predisporre un **dividendo** formato da due cifre **Unpacked BCD**, affinche' una successiva divisione con un **divisore** formato da una cifra **Unpacked BCD**, produca un risultato formalmente identico a quello che si ottiene in base **10**.
In sostanza, **AAD** presuppone che il programmatore stia per eseguire una istruzione **DIV** tra un **dividendo** compreso tra **00h:00h** e **09h:09h**, e un **divisore** compreso tra **01h** e **09h**; in tal caso, **AAD** modifica la struttura del **dividendo**, in modo che **DIV** fornisca un **quoziente** e un **resto**, entrambi in formato **Unpacked BCD**.
In virtu' del fatto che i numeri **BCD** non seguono le leggi dell'aritmetica modulare, e' necessario sottolineare che **AAD** produce un risultato valido solo se viene eseguita prima di una istruzione **DIV** (divisione tra numeri interi senza segno); se, invece, si esegue **AAD** prima di una istruzione **IDIV**, si ottengono risultati privi di senso!

E' importante ribadire che, a differenza di quanto accade con **AAA**, **AAS** e **AAM**, l'istruzione **AAD** deve essere eseguita prima, e non dopo l'operazione da effettuare (che in questo caso e' **DIV**)!
L'unica forma lecita per l'istruzione **AAD**, e' la seguente:

```
AAD
```

Come si puo' notare, il programmatore non deve specificare nessun operando esplicito; infatti, la **CPU** assume che il **dividendo** si trovi nel registro **AX** (operando **SRC**), e utilizza come operando **DEST**, lo stesso registro **AX** (**quoziente** in **AL** e **resto** in **AH**).

Per svolgere il proprio lavoro, l'istruzione **AAD** non fa' altro che compattare in un solo **BYTE**, le due cifre **Unpacked BCD** del **dividendo**; a tale proposito, viene utilizzata una nota proprieta' dei sistemi di numerazione posizionale. Nel caso della base **10**, possiamo scrivere, ad esempio:

$35 = (3 * 10^1) + (5 * 10^0) = (3 * 10) + 5$

Traducendo tutto in formato **Unpacked BCD**, otteniamo:

```
03h:05h = (03h * 10) + 05h = (03h * 0Ah) + 05h = 1Eh + 05h = 23h = 35
```

Una volta che il **dividendo** e' stato compattato, possiamo dividerlo per un **divisore** formato da una sola cifra **Unpacked BCD**, ottenendo cosi' un **quoziente** e un **resto**, entrambi in formato **Unpacked BCD**; vediamo subito alcuni esempi pratici.

Vogliamo calcolare:

```
58 / 7, Q = 8, R = 2
```

Poniamo **AX=AH:AL=05h:08h**, ed eseguiamo l'istruzione:

```
aad
```

In presenza di questa istruzione, la **CPU** calcola:

```
Temp8 = (AH * 10) + AL = (05h * Ah) + 08h = 32h + 08h = 3Ah = 58
```

La **CPU** carica il valore **Temp8=58=3Ah** in **AL**, e azzera il contenuto di **AH** in modo da ottenere **AH:AL=00h:3Ah**; a questo punto, poniamo **BL=07h** ed eseguiamo l'istruzione:

```
div bl
```

In presenza di questa istruzione, la **CPU** calcola:

```
AH:AL = AH:AL / 7 = 58 / 7 = 02h:08h
```

Vogliamo calcolare:

```
35 / 5, Q = 7, R = 0
```

Poniamo **AX=AH:AL=03h:05h**, ed eseguiamo l'istruzione:

```
aad
```

In presenza di questa istruzione, la **CPU** calcola:

```
Temp8 = (AH * 10) + AL = (03h * Ah) + 05h = 1Eh + 05h = 23h = 35
```

La **CPU** carica il valore **Temp8=35=23h** in **AL**, e azzera il contenuto di **AH** in modo da ottenere **AH:AL=00h:23h**; a questo punto, poniamo **BL=05h** ed eseguiamo l'istruzione:

```
div bl
```

In presenza di questa istruzione, la **CPU** calcola:

```
AH:AL = AH:AL / 5 = 35 / 5 = 00h:07h
```

A differenza di quanto accade per **AAA** e **AAS**, l'istruzione **AAD** opera, esclusivamente, su cifre in formato **Unpacked BCD**; non e' possibile quindi utilizzare i codici ASCII dei simboli corrispondenti alle cifre stesse!

18.5.1 Codifica binaria di numeri interi non decimali

L'istruzione **AAD** si serve della base predefinita **10**; esiste pero' la possibilita' di codificare in binario, numeri espressi in qualsiasi altra base. A tale proposito, e' necessario servirsi, obbligatoriamente, di un apposito codice macchina rappresentato dalla sequenza:

```
11010101b Imm8 = D5h Imm8
```

Il valore immediato **Imm8** rappresenta la base da utilizzare (ad esempio, **8**, **8h** o **10q** per la notazione ottale); utilizzando, ad esempio, **Imm8=8**, otteniamo un **quoziente** e un **resto**, entrambi in formato **Unpacked Binary Coded Octal**.

18.5.2 Divisione tra numeri Unpacked BCD di ampiezza arbitraria

Sulla base di quanto e' stato esposto in precedenza, possiamo intuire che la divisione tra numeri **Unpacked BCD** di ampiezza arbitraria, puo' essere svolta in modo semplicissimo; a tale proposito, vediamo subito un esempio pratico.

Supponiamo di voler eseguire la seguente divisione:
`3578 / 9, Q = 397, R = 5`
Dividendo un numero a **4** cifre per un numero a **1** cifra, otteniamo un **quoziente** che richiede, al massimo, **4** cifre, e un **resto** che richiede, al massimo, **1** cifra.
Convertendo tutto in formato **Unpacked BCD**, dobbiamo ottenere:
`03050708h / 09h, Q = 00030907h, R = 05h`
Dividendo un numero da **4 BYTE** per un numero da **1 BYTE**, otteniamo un **quoziente** che richiede, al massimo, **4 BYTE**, e un **resto** che richiede, al massimo, **1 BYTE**.

Come al solito, la divisione inizia con la cifra piu' significativa del **dividendo**; nel nostro caso quindi, il primo quoziente parziale da calcolare, in formato **Unpacked BCD**, e':
`00h:03h / 09h, Q = 00h, R = 03h`
Una istruzione **AAD** produce:
`AH:AL = 00h:03h`
Una successiva istruzione **DIV** con divisore **09h**, produce:
`Q3 = AL = 00h, R3 = AH = 03h`
Il quoziente parziale indicato con **Q3** rappresenta la cifra piu' significativa del **quoziente** finale; al resto parziale **R3** dobbiamo affiancare la seconda cifra (da sinistra) del **dividendo**, in modo da ottenere **03h:05h**.

Il secondo quoziente parziale da calcolare, in formato **Unpacked BCD**, e':
`03h:05h / 9h (Q = 03h, R = 08h)`
Una istruzione **AAD** produce:
`AH:AL = 00h:23h`
Una successiva istruzione **DIV** con divisore **09h**, produce:
`Q2 = AL = 03h, R2 = AH = 08h`
Il quoziente parziale indicato con **Q2** rappresenta la seconda cifra (da sinistra) del **quoziente** finale; al resto parziale **R2** dobbiamo affiancare la terza cifra (da sinistra) del **dividendo**, in modo da ottenere **08h:07h**.

Il terzo quoziente parziale da calcolare, in formato **Unpacked BCD**, e':
`08h:07h / 9h (Q = 09h, R = 06h)`
Una istruzione **AAD** produce:
`AH:AL = 00h:57h`
Una successiva istruzione **DIV** con divisore **09h**, produce:
`Q1 = AL = 09h, R1 = AH = 06h`
Il quoziente parziale indicato con **Q1** rappresenta la terza cifra (da sinistra) del **quoziente** finale; al resto parziale **R1** dobbiamo affiancare la quarta cifra (da sinistra) del **dividendo**, in modo da ottenere **06h:08h**.

Il quarto e ultimo quoziente parziale da calcolare, in formato **Unpacked BCD**, e':
`06h:08h / 9h (Q = 07h, R = 05h)`
Una istruzione **AAD** produce:
`AH:AL = 00h:44h`
Una successiva istruzione **DIV** con divisore **09h**, produce:
`Q0 = AL = 07h, R0 = AH = 05h`
Il quoziente parziale indicato con **Q0** rappresenta cifra meno significativa del **quoziente** finale; il resto parziale **R0** rappresenta il **resto** finale della divisione.

Unendo i **4** quozienti parziali otteniamo il **quoziente** finale **00030907h**; inoltre, abbiamo appena visto che l'ultimo resto parziale **R0**, ci fornisce il **resto** finale **05h**.

Come possiamo notare, il procedimento da seguire e' del tutto simile a quello utilizzato nel precedente capitolo per l'istruzione **DIV**; come al solito, il codice e' molto piu' lungo a causa del fatto che dobbiamo operare sulle singole cifre **Unpacked BCD**.

Se vogliamo implementare in pratica l'esempio precedente, possiamo procedere con la definizione dei dati nel seguente modo:

```
ubcdDividendo  db    8, 7, 5, 3
ubcdDivisore   db    9
ubcdQuoziente  db    4 dup (0)
ubcdResto      db    0
```

In questo caso particolare, il dato **ubcdDividendo** puo' essere definito anche come:
`ubcdDividendo dd 03050708h`
In tal caso, per maneggiare **ubcdDividendo** dobbiamo servirci dell'operatore **BYTE PTR**.

Il codice che esegue la divisione assume il seguente aspetto:

```
; prima divisione con AAD e DIV

mov      ah, 0                  ; ah = 00h
mov      al, ubcdDividendo[3]   ; al = ubcdDividendo[3]
aad                             ; aggiustamento ASCII della divisione
div      ubcdDivisore           ; ah:al = ah:al / ubcdDivisore
mov      ubcdQuoziente[3], al   ; ubcdQuoziente[3] = al

; seconda divisione con AAD e DIV

mov      al, ubcdDividendo[2]   ; al = ubcdDividendo[2]
aad                             ; aggiustamento ASCII della divisione
div      ubcdDivisore           ; ah:al = ah:al / ubcdDivisore
mov      ubcdQuoziente[2], al   ; ubcdQuoziente[2] = al

; terza divisione con AAD e DIV

mov      al, ubcdDividendo[1]   ; al = ubcdDividendo[1]
aad                             ; aggiustamento ASCII della divisione
div      ubcdDivisore           ; ah:al = ah:al / ubcdDivisore
mov      ubcdQuoziente[1], al   ; ubcdQuoziente[1] = al

; quarta divisione con AAD e DIV

mov      al, ubcdDividendo[0]   ; al = ubcdDividendo[0]
aad                             ; aggiustamento ASCII della divisione
div      ubcdDivisore           ; ah:al = ah:al / ubcdDivisore
mov      ubcdQuoziente[0], al   ; ubcdQuoziente[0] = al
mov      ubcdResto, ah          ; ubcdResto = ah
```

Come al solito, e' fondamentale che nella prima divisione, il contenuto di **AH** venga rigorosamente azzerato!

Se vogliamo visualizzare il risultato in formato **Unpacked BCD**, possiamo procedere come nell'esempio su **AAA**; in questo caso, l'output del **quoziente** inizia da **ubcdQuoziente[3]**.

Cosa succede se anche il **divisore** e' formato da due o piu' cifre decimali?
Anche in un caso del genere dobbiamo ricorrere allo stesso algoritmo che si segue con carta e penna; il codice da scrivere diventa pero' notevolmente complesso!
In situazioni del genere, puo' diventare conveniente operare una conversione da **BCD** a binario, eseguire i calcoli in binario, e riconvertire infine i risultati da binario a **BCD**; piu' avanti verranno illustrati ulteriori dettagli su questo aspetto.

In relazione all'eventualita' di dover dividere tra loro numeri **Unpacked BCD** con segno, valgono le stesse considerazioni gia' svolte per **AAM**.

18.5.3 Effetti provocati da AAD sugli operandi e sui flags

L'esecuzione dell'istruzione **AAD** provoca la modifica del solo registro **AX**.

L'esecuzione dell'istruzione **AAD** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; i campi **CF**, **AF**, **OF**, assumono un valore indeterminato, e non hanno quindi nessun significato logico.
I campi **PF**, **ZF** e **SF** forniscono, invece, informazioni ordinarie relative al numero binario memorizzato in **AL** da **AAD**; in sostanza, **PF** indica se **AL** contiene un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il contenuto di **AL** vale zero, **SF** indica se il bit piu' significativo di **AL** vale **1**. Generalmente, anche il contenuto di questi tre flags viene ignorato.

Ovviamente, in relazione all'istruzione **DIV** da eseguire dopo **AAD**, valgono tutte le considerazioni esposte nel precedente capitolo; in presenza quindi di un **dividendo** troppo grande e/o di un **divisore** nullo, si verifica un **overflow di divisione**!

18.6 L'istruzione DAA

Con il mnemonico **DAA** si indica l'istruzione **Decimal adjust AL after Addition** (aggiustamento del contenuto di **AL** dopo una addizione tra due numeri **Packed BCD** a **8** bit); lo scopo di questa istruzione e' quello di aggiustare il risultato di una somma tra due numeri **Packed BCD** a **8** bit, in modo da farlo apparire formalmente identico a quello che si ottiene in base **10**.
In sostanza, **DAA** presuppone che il programmatore abbia appena eseguito una istruzione **ADD** (o **ADC**) tra due numeri **Packed BCD** a **8** bit; in tal caso, **DAA** legge il risultato della somma, e lo converte in formato **Packed BCD** a **8** bit.
L'unica forma lecita per l'istruzione **DAA**, e' la seguente:

```
DAA
```

Come si puo' notare, il programmatore non deve specificare nessun operando esplicito; infatti, la **CPU** assume che la somma si trovi nel registro **AL** (operando **SRC**), e utilizza come operando **DEST**, lo stesso registro **AL**.

L'istruzione **DAA** deve simulare, in formato **Packed BCD**, gli identici risultati che si ottengono sommando due numeri interi decimali, ciascuno compreso tra **0** e **99**; di conseguenza, **DAA** deve anche simulare un eventuale riporto dal nibble basso al nibble alto (**AF**), e un eventuale riporto al byte successivo (**CF**).
Per capire il principio di funzionamento di **DAA**, osserviamo innanzi tutto che la somma tra numeri

decimali a due cifre, e' sempre compresa tra il valore minimo:
`0 + 0 = 0`
e il valore massimo:
`99 + 99 = 198` (cioe', **98** con riporto di **1**).
Si possono presentare allora diverse possibilita', che vengono segnalate dalla **CPU** attraverso la modifica (separata) dei flags **AF** e **CF**; a tale proposito, analizziamo alcuni esempi pratici.

Vogliamo calcolare:
`2 + 6 = 8`
Poniamo **AL=02h**, **BL=06h**, ed eseguiamo l'istruzione:
`add al, bl`
In presenza di questa istruzione, la **CPU** calcola:
`AL = AL + BL = 02h + 06h = 00000010b + 00000110b = 00001000b = 08h, con AF = 0, CF = 0`
A questo punto, dobbiamo eseguire l'istruzione:
`daa`
L'istruzione **ADD** ha prodotto **AF=0** (nessun riporto dal nibble basso al nibble alto), e inoltre, la cifra **8h** contenuta nel nibble basso di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**AF** e **CF** rimangono a **0**).
I calcoli precedenti hanno prodotto **CF=0** (nessun riporto al byte successivo), e inoltre, la cifra **0h** contenuta nel nibble alto di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**CF** rimane a **0**).
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=0** e **CF=0**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=08h**.

Vogliamo calcolare:
`5 + 9 = 14`
Poniamo **AL=05h**, **BL=09h**, ed eseguiamo l'istruzione:
`add al, bl`
In presenza di questa istruzione, la **CPU** calcola:
`AL = AL + BL = 05h + 09h = 00000101b + 00001001b = 00001110b = 0Eh, con AF = 0, CF = 0`
A questo punto, dobbiamo eseguire l'istruzione:
`daa`
L'istruzione **ADD** ha prodotto **AF=0** (nessun riporto dal nibble basso al nibble alto), ma la cifra **Eh** contenuta nel nibble basso di **AL**, supera **9h**; di conseguenza, tale cifra necessita di un opportuno aggiustamento.
La **CPU** somma **06h** al contenuto di **AL**, e ottiene:
`AL = AL + 06h = 0Eh + 06h = 14h, con TempCF = 0`
Inoltre, la **CPU** pone:
`AF = 1` e `CF = CF OR TempCF = 0 OR 0 = 0`
I calcoli precedenti hanno prodotto **CF=0** (nessun riporto al byte successivo), e inoltre, la cifra **1h** contenuta nel nibble alto di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**CF** rimane a **0**).
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=1** e **CF=0**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=14h**.

Vogliamo calcolare:
`95 + 10 = 105`
Poniamo **AL=95h**, **BL=10h**, ed eseguiamo l'istruzione:
`add al, bl`
In presenza di questa istruzione, la **CPU** calcola:
`AL = AL + BL = 95h + 10h = 10010101b + 00010000b = 10100101b = A5h, con AF = 0, CF = 0`

A questo punto, dobbiamo eseguire l'istruzione:
```
daa
```
L'istruzione **ADD** ha prodotto **AF=0** (nessun riporto dal nibble basso al nibble alto), e inoltre, la cifra **5h** contenuta nel nibble basso di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**AF** e **CF** rimangono a **0**).
I calcoli precedenti hanno prodotto **CF=0** (nessun riporto al byte successivo), ma la cifra **Ah** contenuta nel nibble alto di **AL**, supera **9h**; di conseguenza, tale cifra necessita di un aggiustamento.
La **CPU** somma **60h** al contenuto di **AL**, e ottiene:
```
AL = AL + 60h = A5h + 60h = 05h
```
Inoltre, la **CPU** pone:
```
CF = 1
```
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=0** e **CF=1**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=05h**.

Vogliamo calcolare:
```
89 + 79 = 168
```
Poniamo **AL=89h**, **BL=79h**, ed eseguiamo l'istruzione:
```
add al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL + BL = 89h + 79h = 10001001b + 01111001b = 00000010b = 02h, con AF = 1,
CF = 1
```
A questo punto, dobbiamo eseguire l'istruzione:
```
daa
```
L'istruzione **ADD** ha prodotto **AF=1** (riporto dal nibble basso al nibble alto); di conseguenza, la cifra contenuta nel nibble basso di **AL** necessita sicuramente di un aggiustamento.
La **CPU** somma **06h** al contenuto di **AL**, e ottiene:
```
AL = AL + 06h = 02h + 06h = 08h, con TempCF = 0
```
Inoltre, la **CPU** pone:
`AF = 1` e `CF = CF OR TempCF = 1 OR 0 = 1`
I calcoli precedenti hanno prodotto **CF=1** (riporto al byte successivo); di conseguenza, la cifra contenuta nel nibble alto di **AL** necessita sicuramente di un aggiustamento.
La **CPU** somma **60h** al contenuto di **AL**, e ottiene:
```
AL = AL + 60h = 08h + 60h = 68h
```
Inoltre, la **CPU** pone:
```
CF = 1
```
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=1** e **CF=1**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=68h**.

Riassumendo, la **CPU** utilizza **AF** per indicarci se la somma **Packed BCD** ha prodotto un riporto dal nibble basso al nibble alto; analogamente, la **CPU** utilizza **CF** per indicarci se la somma **Packed BCD** ha prodotto un riporto verso un ipotetico gruppo successivo di cifre **Packed BCD**. Il risultato della somma, memorizzato in **AL**, e' identico a quello che si ottiene in base **10**; inoltre, attraverso **CF**, possiamo sommare numeri **Packed BCD** di ampiezza arbitraria.

18.6.1 Somma tra numeri Packed BCD di ampiezza arbitraria

Sulla base di quanto e' stato esposto in precedenza, possiamo intuire che la somma tra numeri **Packed BCD** di ampiezza arbitraria, puo' essere svolta in modo ancora piu' semplice rispetto al caso dei numeri **Unpacked BCD**; in relazione all'esempio presentato per l'istruzione **AAA**, le uniche differenze sono le seguenti:

***** con i numeri **Packed BCD** operiamo su due cifre per volta;
***** al posto dell'istruzione **AAA** dobbiamo utilizzare **DAA**.

Vediamo subito un esempio pratico che ci permette di illustrare l'utilizzo dei numeri **Packed BCD** standard; a tale proposito, supponiamo di voler eseguire la seguente somma tra numeri interi decimali da **18** cifre ciascuno:

```
(+743251687690345672) +
(+215467893427034538) =
-----------------------
 +958719581117380210
```

Convertendo tutto in formato **Packed BCD** standard, dobbiamo ottenere:

```
00743251687690345672h +
00215467893427034538h =
-----------------------
00958719581117380210h
```

Ricordiamo che, il segno di un numero **Packed BCD** standard, viene codificato nel **BYTE** piu' significativo; il codice **00h** rappresenta il segno positivo, mentre il codice **80h** rappresenta il segno negativo.

Per definire i vari dati da elaborare, possiamo servirci della direttiva **DT** (**Define Tenbyte**) nel seguente modo:

```
pbcdNum1    dt    00743251687690345672h   ; +743251687690345672
pbcdNum2    dt    00215467893427034538h   ; +215467893427034538
pbcdSomma   dt    0h
```

Il codice che esegue la somma assume il seguente aspetto:

```
; somma dei primi BYTE con ADD e DAA

mov      al, byte ptr pbcdNum1[0]   ; al = pbcdNum1[0]
add      al, byte ptr pbcdNum2[0]   ; al = pbcdNum1[0] + pbcdNum2[0]
daa                                 ; aggiustamento decimale dell'addizione
mov      byte ptr pbcdSomma[0], al  ; pbcdSomma[0] = al

; somma dei secondi BYTE con ADC e DAA

mov      al, byte ptr pbcdNum1[1]   ; al = pbcdNum1[1]
adc      al, byte ptr pbcdNum2[1]   ; al = pbcdNum1[1] + pbcdNum2[1] + CF
daa                                 ; aggiustamento decimale dell'addizione
mov      byte ptr pbcdSomma[1], al  ; pbcdSomma[1] = al

; somma dei terzi BYTE con ADC e DAA

mov      al, byte ptr pbcdNum1[2]   ; al = pbcdNum1[2]
adc      al, byte ptr pbcdNum2[2]   ; al = pbcdNum1[2] + pbcdNum2[2] + CF
daa                                 ; aggiustamento decimale dell'addizione
mov      byte ptr pbcdSomma[2], al  ; pbcdSomma[2] = al

; ... e cosi' via per gli altri 6 BYTE

mov      byte ptr pbcdSomma[9], 00h ; risultato con segno positivo

call     showCPU                    ; verifica dei flags
```

Dopo l'esecuzione dell'ultima somma (tra gli ottavi **BYTE**), consultiamo **CF** per sapere se c'e' stato un riporto finale; nel nostro caso, **CF=0**, per cui il risultato e' valido cosi' com'e'.
Osserviamo che, sommando due numeri positivi, otteniamo un risultato positivo; di conseguenza, dobbiamo memorizzare il valore **00h** nel **BYTE** piu' significativo del risultato.

Se vogliamo visualizzare il risultato in formato **Packed BCD**, possiamo utilizzare lo stesso procedimento illustrato per **AAA**; in questo caso, l'output inizia da **pbcdSomma[9]**.

18.6.2 Effetti provocati da DAA sugli operandi e sui flags

L'esecuzione dell'istruzione **DAA** provoca la modifica del solo registro **AL**.

L'esecuzione dell'istruzione **DAA** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **OF** assume un valore indeterminato, e non ha quindi nessun significato.
I campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al numero binario memorizzato in **AL** da **DAA**; in sostanza, **PF** indica se **AL** contiene un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il contenuto di **AL** vale zero, **SF** indica se il bit piu' significativo di **AL** vale **1**.
Generalmente, anche il contenuto di questi tre flags viene ignorato.
Se la somma **Packed BCD** provoca un riporto dal nibble basso al nibble alto, la **CPU** pone **AF=1**; in caso contrario si ha **AF=0**. Se la somma **Packed BCD** provoca un riporto al byte successivo, la **CPU** pone **CF=1**; in caso contrario si ha **CF=0**.

18.7 L'istruzione DAS

Con il mnemonico **DAS** si indica l'istruzione **Decimal adjust AL after Subtraction** (aggiustamento del contenuto di **AL** dopo una sottrazione tra due numeri **Packed BCD** a **8** bit); lo scopo di questa istruzione e' quello di aggiustare il risultato di una differenza tra due numeri **Packed BCD** a **8** bit, in modo da farlo apparire formalmente identico a quello che si ottiene in base **10**.
In sostanza, **DAS** presuppone che il programmatore abbia appena eseguito una istruzione **SUB** (o **SBB**) tra due numeri **Packed BCD** a **8** bit; in tal caso, **DAS** legge il risultato della differenza, e lo converte in formato **Packed BCD** a **8** bit.
L'unica forma lecita per l'istruzione **DAS**, e' la seguente:

```
DAS
```

Come si puo' notare, il programmatore non deve specificare nessun operando esplicito; infatti, la **CPU** assume che la differenza si trovi nel registro **AL** (operando **SRC**), e utilizza come operando **DEST**, lo stesso registro **AL**.

L'istruzione **DAS** deve simulare, in formato **Packed BCD**, gli identici risultati che si ottengono sottraendo due numeri interi decimali, ciascuno compreso tra **0** e **99**; di conseguenza, **DAS** deve anche simulare un eventuale prestito dal nibble alto al nibble basso (**AF**), e un eventuale prestito dal byte successivo (**CF**).

Si possono presentare allora diverse possibilita', che vengono segnalate dalla **CPU** attraverso la modifica (separata) dei flags **AF** e **CF**; a tale proposito, analizziamo alcuni esempi pratici.

Vogliamo calcolare:
```
37 - 12 = 25
```

Poniamo **AL=37h**, **BL=12h**, ed eseguiamo l'istruzione:
```
sub al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL - BL = 37h - 12h = 00110111b - 00010010b = 00100101b = 25h, con AF = 0,
CF = 0
```
A questo punto, dobbiamo eseguire l'istruzione:
```
das
```
L'istruzione **SUB** ha prodotto **AF=0** (nessun prestito dal nibble alto al nibble basso), e inoltre, la cifra **5h** contenuta nel nibble basso di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**AF** e **CF** rimangono a **0**).
I calcoli precedenti hanno prodotto **CF=0** (nessun prestito dal byte successivo), e inoltre, la cifra **2h** contenuta nel nibble alto di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**CF** rimane a **0**).
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=0** e **CF=0**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=25h**.

Vogliamo calcolare:
```
45 - 18 = 27
```
Poniamo **AL=45h**, **BL=18h**, ed eseguiamo l'istruzione:
```
sub al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL - BL = 45h - 18h = 01000101b - 00011000b = 00101101b = 2Dh, con AF = 1,
CF = 0
```
A questo punto, dobbiamo eseguire l'istruzione:
```
das
```
L'istruzione **SUB** ha prodotto **AF=1** (prestito dal nibble alto al nibble basso); di conseguenza, la cifra contenuta nel nibble basso di **AL** necessita sicuramente di un aggiustamento.
La **CPU** sottrae **06h** al contenuto di **AL**, e ottiene:
```
AL = AL - 06h = 2Dh - 06h = 27h, con TempCF = 0
```
Inoltre, la **CPU** pone:
`AF = 1` e `CF = CF OR TempCF = 0 OR 0 = 0`
I calcoli precedenti hanno prodotto **CF=0** (nessun prestito dal byte successivo), e inoltre, la cifra **2h** contenuta nel nibble alto di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**CF** rimane a **0**).
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=1** e **CF=0**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=27h**.

Vogliamo calcolare:
`38 - 71 = 67` (con prestito di **1**).
Poniamo **AL=38h**, **BL=71h**, ed eseguiamo l'istruzione:
```
sub al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL - BL = 38h - 71h = 00111000b - 01110001b = 11000111b = C7h, con AF = 0,
CF = 1
```
A questo punto, dobbiamo eseguire l'istruzione:
```
daa
```
L'istruzione **SUB** ha prodotto **AF=0** (nessun prestito dal nibble alto al nibble basso), e inoltre, la cifra **7h** contenuta nel nibble basso di **AL**, non supera **9h**; di conseguenza, tale cifra non necessita di nessun aggiustamento (**AF** rimane a **0** e **CF** rimane a **1**).
I calcoli precedenti hanno prodotto **CF=1** (prestito dal byte successivo); di conseguenza, la cifra contenuta nel nibble alto di **AL** necessita sicuramente di un aggiustamento.
La **CPU** sottrae **60h** al contenuto di **AL**, e ottiene:
```
AL = AL - 60h = C7h - 60h = 67h
```
Inoltre, la **CPU** pone:

`CF = 1`
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=0** e **CF=1**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=67h**.

Vogliamo calcolare:
`31 - 79 = 52` (con prestito di **1**).
Poniamo **AL=31h**, **BL=79h**, ed eseguiamo l'istruzione:
```
sub al, bl
```
In presenza di questa istruzione, la **CPU** calcola:
```
AL = AL - BL = 31h - 79h = 00110001b - 01111001b = 10111000b = B8h, con AF = 1,
CF = 1
```
A questo punto, dobbiamo eseguire l'istruzione:
```
das
```
L'istruzione **SUB** ha prodotto **AF=1** (prestito dal nibble alto al nibble basso); di conseguenza, la cifra contenuta nel nibble basso di **AL** necessita sicuramente di un aggiustamento.
La **CPU** sottrae **06h** al contenuto di **AL**, e ottiene:
```
AL = AL - 06h = B8h - 06h = B2h, con TempCF = 0
```
Inoltre, la **CPU** pone:
`AF = 1` e `CF = CF OR TempCF = 1 OR 0 = 1`
I calcoli precedenti hanno prodotto **CF=1** (prestito dal byte successivo); di conseguenza, la cifra contenuta nel nibble alto di **AL** necessita sicuramente di un aggiustamento.
La **CPU** sottrae **60h** al contenuto di **AL**, e ottiene:
```
AL = AL - 60h = B2h - 60h = 52h
```
Inoltre, la **CPU** pone:
```
CF = 1
```
Le elaborazioni eseguite dalla **CPU** hanno prodotto **AF=1** e **CF=1**; alla fine otteniamo il risultato **Packed BCD** dato da **AL=52h**.

Riassumendo, la **CPU** utilizza **AF** per indicarci se la differenza **Packed BCD** ha prodotto un prestito dal nibble alto al nibble basso; analogamente, la **CPU** utilizza **CF** per indicarci se la differenza **Packed BCD** ha prodotto un prestito da un ipotetico gruppo successivo di cifre **Packed BCD**. Il risultato della differenza, memorizzato in **AL**, e' identico a quello che si ottiene in base **10**; inoltre, attraverso **CF**, possiamo sottrarre numeri **Packed BCD** di ampiezza arbitraria.

18.7.1 Differenza tra numeri Packed BCD di ampiezza arbitraria

Sulla base di quanto e' stato esposto in precedenza, possiamo intuire che la differenza tra numeri **Packed BCD** di ampiezza arbitraria, puo' essere svolta in modo ancora piu' semplice rispetto al caso dei numeri **Unpacked BCD**; in relazione all'esempio presentato per l'istruzione **AAS**, le uniche differenze sono le seguenti:

***** con i numeri **Packed BCD** operiamo su due cifre per volta;
***** al posto dell'istruzione **AAS** dobbiamo utilizzare **DAS**.

Vediamo subito un esempio pratico che ci permette di illustrare l'utilizzo dei numeri **Packed BCD** standard; a tale proposito, supponiamo di voler eseguire la seguente somma tra numeri interi decimali da **18** cifre ciascuno:

```
(-743251687690345672) +
(+215467893427034538) =
-----------------------
 -527783794263311134
```

Convertendo tutto in formato **Packed BCD** standard, dobbiamo ottenere:

```
80743251687690345672h +
00215467893427034538h =
----------------------
80527783794263311134h
```

Osserviamo subito che il primo numero e' negativo (**80h**), mentre il secondo e' positivo (**00h**); di conseguenza, il procedimento da seguire consiste nel calcolare una differenza tra numeri positivi, assegnando poi il segno negativo (**80h**) al risultato.

Per definire i vari dati da elaborare, possiamo servirci della direttiva **DT** (**Define Tenbyte**) nel seguente modo:

```
pbcdNum1    dt    80743251687690345672h   ; -743251687690345672
pbcdNum2    dt    00215467893427034538h   ; +215467893427034538
pbcdDiff    dt    0h
```

Il codice che esegue la differenza assume il seguente aspetto:

```
; differenza tra i primi BYTE con SUB e DAS

mov      al, byte ptr pbcdNum1[0]   ; al = pbcdNum1[0]
sub      al, byte ptr pbcdNum2[0]   ; al = pbcdNum1[0] - pbcdNum2[0]
das                                 ; aggiustamento decimale della sottrazione
mov      byte ptr pbcdDiff[0], al   ; pbcdDiff[0] = al

; differenza tra i secondi BYTE con SBB e DAS

mov      al, byte ptr pbcdNum1[1]   ; al = pbcdNum1[1]
sbb      al, byte ptr pbcdNum2[1]   ; al = pbcdNum1[1] - pbcdNum2[1] - CF
das                                 ; aggiustamento decimale della sottrazione
mov      byte ptr pbcdDiff[1], al   ; pbcdDiff[1] = al

; differenza tra i terzi BYTE con SBB e DAS

mov      al, byte ptr pbcdNum1[2]   ; al = pbcdNum1[2]
sbb      al, byte ptr pbcdNum2[2]   ; al = pbcdNum1[2] - pbcdNum2[2] - CF
das                                 ; aggiustamento decimale della sottrazione
mov      byte ptr pbcdDiff[2], al   ; pbcdDiff[2] = al

; ... e cosi' via per gli altri 6 BYTE

mov      byte ptr pbcdDiff[9], 80h  ; risultato con segno negativo

call     showCPU                    ; verifica dei flags
```

Dopo l'esecuzione dell'ultima differenza (tra gli ottavi **BYTE**), consultiamo **CF** per sapere se c'e' stato un prestito finale; nel nostro caso, **CF=0**, per cui il risultato e' valido cosi' com'e'.

Se vogliamo visualizzare il risultato in formato **Packed BCD**, possiamo utilizzare lo stesso procedimento illustrato per **AAA**; in questo caso, l'output inizia da **pbcdDiff[9]**.

18.7.2 Effetti provocati da DAS sugli operandi e sui flags

L'esecuzione dell'istruzione **DAS** provoca la modifica del solo registro **AL**.

L'esecuzione dell'istruzione **DAS** provoca la modifica dei campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **OF** assume un valore indeterminato, e non ha quindi nessun significato.
I campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al numero binario memorizzato in **AL** da **DAS**; in sostanza, **PF** indica se **AL** contiene un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il contenuto di **AL** vale zero, **SF** indica se il bit piu' significativo di **AL** vale **1**.
Generalmente, anche il contenuto di questi tre flags viene ignorato.
Se la differenza **Packed BCD** provoca un prestito dal nibble alto al nibble basso, la **CPU** pone **AF=1**; in caso contrario si ha **AF=0**. Se la differenza **Packed BCD** provoca un prestito dal byte successivo, la **CPU** pone **CF=1**; in caso contrario si ha **CF=0**.

18.8 Conversione da Packed BCD a Unpacked BCD e viceversa

Le **CPU** della famiglia **80x86**, non dispongono di istruzioni per l'aggiustamento decimale di moltiplicazioni e divisioni; per i numeri **Packed BCD** quindi, non esiste nessuna istruzione equivalente a **AAM** e **AAD**. Eventualmente, il programmatore puo' scrivere apposite procedure che svolgono questo tipo di operazioni; a tale proposito, e' necessario servirsi degli stessi concetti su cui si basa il principio di funzionamento delle istruzioni **AAM** e **AAD**.
In alternativa, si puo' anche seguire un'altra strada, che consiste nel convertire i numeri **BCD**, da un formato all'altro; volendo eseguire, ad esempio, una moltiplicazione tra numeri **Packed BCD** di ampiezza arbitraria, possiamo pensare di convertire i due fattori, da **Packed BCD** a **Unpacked BCD**, svolgere i calcoli con **MUL** e **AAM**, e poi convertire il risultato da **Unpacked BCD** a **Packed BCD**.

18.8.1 Conversione da Packed BCD a Unpacked BCD

La conversione da **Packed BCD** a **Unpacked BCD** si svolge in modo semplicissimo grazie all'istruzione **AAM**; infatti, sappiamo che **AAM** legge da **AL** il risultato di una moltiplicazione tra fattori **Unpacked BCD**, lo converte in formato **Unpacked BCD** e lo memorizza in **AH:AL**.
Consideriamo, ad esempio, il seguente prodotto:
`AL = 03h * 09h = 1Bh = 27`
Una successiva istruzione **AAM** calcola, come sappiamo:
`1Bh / Ah, AH = Q = 02h, AL = R = 07h`
L'istruzione **AAM** utilizza la base predefinita **10**, e proprio per questo motivo, ottiene un risultato decimale; il trucco da utilizzare, consiste allora nel servirsi di **AAM** con base **16**!
Come gia' sappiamo, per imporre a **AAM** l'utilizzo della base **16**, dobbiamo servirci del codice macchina **D4h, 16**; nel caso dell'esempio precedente, otteniamo allora:
`1Bh / 10h, AH = Q = 01h, AL = R = 0Bh`
Come si puo' notare, abbiamo convertito **1Bh** in **01h:0Bh**!

Una volta chiarito questo trucco, possiamo passare ad un esempio pratico; a tale proposito, supponiamo di voler convertire in formato **Unpacked BCD**, il numero **Packed BCD** standard **80743251687690345672h**.
Come sappiamo, la versione **Unpacked BCD** di un numero, occupa il doppio dei byte rispetto alla versione **Packed BCD**; di conseguenza, possiamo procedere con le seguenti definizioni:

```
pbcdNumber dt    80743251687690345672h   ; -743251687690345672
ubcdNumber db    20 dup (0)
```

Il seguente codice esegue la conversione da **Packed BCD** standard a **Unpacked BCD**:

```
; inizializzazione puntatori

mov     si, offset pbcdNumber      ; ds:si punta a pbcdNumber
mov     di, offset ubcdNumber      ; ds:di punta a ubcdNumber

; conversione del primo BYTE di pbcdNumber

mov     al, [si]                   ; al = pbcdNumber[0]
db      0D4h, 16                   ; AAM con base 16
mov     [di], ax                   ; ubcdNumber[0] = al, ubcdNumber[1] = ah
inc     si                         ; prossimo BYTE di pbcdNumber
add     di, 2                      ; prossima WORD di ubcdNumber

; conversione del secondo BYTE di pbcdNumber

mov     al, [si]                   ; al = pbcdNumber[1]
db      0D4h, 16                   ; AAM con base 16
mov     [di], ax                   ; ubcdNumber[2] = al, ubcdNumber[3] = ah
inc     si                         ; prossimo BYTE di pbcdNumber
add     di, 2                      ; prossima WORD di ubcdNumber

; ... e cosi' via per gli altri 8 BYTE di pbcdNumber
```

Per ogni conversione, il puntatore (**SI**) a **pbcdNumber** deve essere incrementato di **1** byte, mentre il puntatore (**DI**) a **ubcdNumber** deve essere incrementato di **2** byte; infatti, ogni **BYTE** di **pbcdNumber** (che contiene una coppia di cifre in formato **Packed BCD**), deve essere convertito in una **WORD** di **ubcdNumber** (che contiene una coppia di cifre in formato **Unpacked BCD**).
Osserviamo che scompattando anche il segno **80h** di **pbcdNumber**, otteniamo **0800h**; il programmatore deve quindi tenere presente che la **WORD** piu' significativa di **ubcdNumber**, contiene la codifica del segno del numero appena convertito.

Se vogliamo visualizzare il risultato in formato **Unpacked BCD**, possiamo utilizzare lo stesso procedimento illustrato per **AAA**; in questo caso, l'output inizia da **ubcdNumber[19]**.

18.8.2 Conversione da Unpacked BCD a Packed BCD

La conversione da **Unpacked BCD** a **Packed BCD** si svolge in modo semplicissimo grazie all'istruzione **AAD**; infatti, sappiamo che **AAD** prepara il contenuto **Unpacked BCD** della coppia **AH:AL**, in modo che una successiva istruzione **DIV** fornisca un **quoziente** e un **resto**, entrambi in formato **Unpacked BCD**.
Supponiamo, ad esempio, di avere **AH:AL=01h:08h**; una successiva istruzione **AAD** calcola, come sappiamo:
`(01h * Ah) + 08h = 0Ah + 08h = 12h = 18`
L'istruzione **AAD** utilizza la base predefinita **10**, e proprio per questo motivo, ottiene un risultato decimale; il trucco da utilizzare, consiste allora nel servirsi di **AAD** con base **16**!
Come gia' sappiamo, per imporre a **AAD** l'utilizzo della base **16**, dobbiamo servirci del codice macchina **D5h, 16**; nel caso dell'esempio precedente, otteniamo allora:
`(01h * 10h) + 08h = 10h + 08h = 18h`
Come si puo' notare, abbiamo convertito **01h:08h** in **18h**!

Una volta chiarito questo trucco, possiamo passare ad un esempio pratico; a tale proposito, supponiamo di voler convertire in formato **Packed BCD** standard, il numero **Unpacked BCD** dell'esempio precedente, rappresentato da **08000704030205010608070609000304050607 02h**.

Come sappiamo, la versione **Packed BCD** standard di un numero, occupa la meta' dei byte rispetto alla versione **Unpacked BCD**; di conseguenza, possiamo procedere con le seguenti definizioni:

```
ubcdNumber db    2, 7, 6, 5, 4, 3, 0, 9, 6, 7, 8, 6, 1, 5, 2, 3, 4, 7, 0, 8
pbcdNumber dt    0h
```

Il seguente codice esegue la conversione da **Unpacked BCD** a **Packed BCD** standard:

```
; inizializzazione puntatori

mov      si, offset ubcdNumber      ; ds:si punta a ubcdNumber
mov      di, offset pbcdNumber      ; ds:di punta a pbcdNumber

; conversione della prima WORD di ubcdNumber

mov      ax, [si]                   ; al = ubcdNumber[0], ah = ubcdNumber[1]
db       0D5h, 16                   ; AAD con base 16
mov      [di], al                   ; pbcdNumber[0] = al
add      si, 2                      ; prossima WORD di ubcdNumber
inc      di                         ; prossimo BYTE di pbcdNumber

; conversione della seconda WORD di ubcdNumber

mov      ax, [si]                   ; al = ubcdNumber[2], ah = ubcdNumber[3]
db       0D5h, 16                   ; AAD con base 16
mov      [di], al                   ; pbcdNumber[1] = al
add      si, 2                      ; prossima WORD di ubcdNumber
inc      di                         ; prossimo BYTE di pbcdNumber

; ... e cosi' via per le altre 8 WORD di ubcdNumber
```

Per ogni conversione, il puntatore (**SI**) a **ubcdNumber** deve essere incrementato di **2** byte, mentre il puntatore (**DI**) a **pbcdNumber** deve essere incrementato di **1** byte; infatti, ogni **WORD** di **ubcdNumber** (che contiene una coppia di cifre in formato **Unpacked BCD**), deve essere convertita in un **BYTE** di **pbcdNumber** (che contiene una coppia di cifre in formato **Packed BCD**).

Se vogliamo visualizzare il risultato in formato **Packed BCD**, possiamo utilizzare lo stesso procedimento illustrato per **AAA**; in questo caso, l'output inizia da **pbcdNumber[9]**.

18.9 Conversione da BCD a binario e viceversa

Quando le elaborazioni da svolgere sui numeri **BCD** diventano troppo complesse, potrebbe risultare conveniente una conversione in binario dei numeri stessi; in questo modo, si effettuano le varie elaborazioni sui numeri binari, e si provvede poi a riconvertire in formato **BCD** i risultati appena ottenuti.
Per effettuare le conversioni da **BCD** a binario e viceversa, possiamo ricorrere, ad esempio, al **metodo delle divisioni successive**; come e' stato gia' ricordato anche in questo capitolo, dividendo ripetutamente un numero in base **b1**, per una base **b2**, si ottengono una serie di resti che formano la sequenza delle cifre, in base **b2**, del numero stesso. Le divisioni terminano non appena si ottiene un **quoziente** nullo; l'ultimo resto ottenuto rappresenta la cifra piu' significativa del numero appena convertito in base **b2**.

Il problema fondamentale consiste nel fatto che dobbiamo cercare di ricondurci sempre al caso di una divisione tra un **dividendo** di ampiezza arbitraria, e un **divisore** formato da una solo cifra (che rappresenta la base di destinazione); in questo modo, possiamo applicare gli stessi procedimenti gia'

illustrati in questo capitolo e nel capitolo precedente.

18.9.1 Conversione da BCD a binario

Per la conversione da **BCD** a binario, dobbiamo utilizzare la cifra **2** (base dei numeri binari) come **divisore**; in questo modo, e' possibile applicare lo stesso procedimento gia' illustrato in questo capitolo per l'istruzione **AAD** (ovviamente, se il **dividendo** e' in formato **Packed BCD**, dobbiamo prima convertirlo in formato **Unpacked BCD**).

Supponiamo, ad esempio, di voler convertire in binario il numero decimale **07050301h** (in formato **Unpacked BCD**); dividendo ripetutamente tale numero per **02h** con l'ausilio di **AAD** e **DIV**, otteniamo:

```
07050301h / 02h, Q = 03070605h, R = 01h

03070605h / 02h, Q = 01080802h, R = 01h

01080802h / 02h, Q = 00090401h, R = 00h

00090401h / 02h, Q = 00040700h, R = 01h

00040700h / 02h, Q = 00020305h, R = 00h

00020305h / 02h, Q = 00010107h, R = 01h

00010107h / 02h, Q = 00000508h, R = 01h

00000508h / 02h, Q = 00000209h, R = 00h

00000209h / 02h, Q = 00000104h, R = 01h

00000104h / 02h, Q = 00000007h, R = 00h

00000007h / 02h, Q = 00000003h, R = 01h

00000003h / 02h, Q = 00000001h, R = 01h

00000001h / 02h, Q = 00000000h, R = 01h
```

Unendo assieme i vari resti (a partire dall'ultimo), otteniamo il numero binario a **13** cifre **1110101101011b**; come verifica, osserviamo che traducendo tale numero in base **10**, risulta:

$$(1 * 2^{12}) + (1 * 2^{11}) + (1 * 2^{10}) + (0 * 2^{9}) + (1 * 2^{8}) + (0 * 2^{7}) + (1 * 2^{6}) + (1 * 2^{5}) + (0 * 2^{4}) + (1 * 2^{3}) + (0 * 2^{2}) + (1 * 2^{1}) + (1 * 2^{0}) =$$

$$4096 + 2048 + 1024 + 0 + 256 + 0 + 64 + 32 + 0 + 8 + 0 + 2 + 1 = 7531$$

Per poter memorizzare i vari resti, nei singoli bit di un numero binario, abbiamo bisogno delle istruzioni logiche; tali istruzioni, vengono illustrate nel capitolo successivo.

18.9.2 Conversione da binario a BCD

Per la conversione da binario a **BCD**, dobbiamo utilizzare la cifra **Ah** (base dei numeri decimali) come **divisore**; in questo modo, e' possibile applicare lo stesso procedimento gia' illustrato nel capitolo precedente per l'istruzione **DIV**.

Supponiamo, ad esempio, di voler convertire in **BCD** il numero binario **1110101101011b=1D6Bh** che abbiamo ottenuto in precedenza; dividendo ripetutamente tale numero per **Ah** con l'ausilio di **DIV**, otteniamo:

```
1DB6h / Ah, Q = 02F1h, R = 1h

02F1h / Ah, Q = 004Bh, R = 3h

004Bh / Ah, Q = 0007h, R = 5h

0007h / Ah, Q = 0000h, R = 7h
```

Unendo assieme i vari resti (a partire dall'ultimo), otteniamo il numero decimale a **4** cifre **7531**; avendo a disposizione le singole cifre del numero, possiamo facilmente memorizzarle in formato **Unpacked BCD** (per il formato **Packed BCD** abbiamo bisogno delle istruzioni logiche).
Nota importante.
In base alle considerazioni esposte in questo capitolo, risulta evidente il fatto che, le varie istruzioni per il formato **BCD**, presuppongono di operare su numeri codificati correttamente; cio' significa che se applichiamo queste istruzioni a numeri del tipo **FCh**, **0Dh**, **ACh**, etc, otteniamo risultati privi di senso!

Capitolo 19 - Istruzioni logiche

Nei precedenti capitoli, abbiamo lasciato in sospeso alcune questioni che non potevano essere affrontate con l'ausilio delle sole istruzioni che gia' conosciamo (istruzioni aritmetiche e di trasferimento dati); tali questioni riguardavano, principalmente, la necessita' di poter operare sui singoli bit di un numero binario.

Abbiamo visto, ad esempio, che moltiplicando tra loro due numeri interi di ampiezza arbitraria, si ottengono una serie di prodotti parziali, che devono essere poi sommati tra loro in modo da ottenere il prodotto finale; prima di poter effettuare tale somma, sappiamo pero' che e' necessario sottoporre a scorrimento verso sinistra, le singole cifre dei prodotti parziali stessi. Come si effettua una tale operazione?

Nel precedente capitolo, invece, abbiamo visto che per convertire in binario un numero **Unpacked BCD**, possiamo servirci del metodo delle divisioni successive, con divisore **2**; in questo modo, otteniamo una sequenza di resti, ciascuno dei quali rappresenta una delle cifre del numero appena convertito in binario. Come facciamo a memorizzare tali cifre, nei singoli bit di un numero binario?

Per poter rispondere a queste domande, dobbiamo studiare in dettaglio una importante categoria di istruzioni della **CPU**, che vengono definite: **istruzioni logiche**; si tratta di istruzioni estremamente potenti, che permettono di effettuare, ad esempio, scorrimenti, rotazioni, test, inversioni, e numerose altre operazioni, sui singoli bit di un numero binario.
Grazie a queste particolari caratteristiche, le istruzioni logiche vengono largamente impiegate dai programmatori **Assembly**, per realizzare sofisticati trucchi di programmazione; a tale proposito, in questo capitolo vedremo numerosi esempi pratici.

Appare intuitivo il fatto che il principio di funzionamento delle istruzioni logiche, e' strettamente legato ai concetti esposti nei capitoli 3, 4, 5 e 6; eventualmente, una rapida rilettura di tali capitoli, puo' aiutare a capire meglio cio' che verra' esposto nel seguito.

Come accade per le istruzioni aritmetiche, anche per le istruzioni logiche e' proibito l'uso di operandi di tipo **SegReg**; nel seguito del capitolo quindi, quando si parla di generici registri, ci si riferisce ai registri generali della **CPU**.

19.1 L'istruzione AND

Con il mnemonico **AND** si indica l'istruzione **logical AND** (**AND** logico); lo scopo di questa istruzione e' quello di effettuare un **AND** logico tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
AND   Reg, Reg
AND   Reg, Mem
AND   Mem, Reg
AND   Reg, Imm
AND   Mem, Imm
```

L'istruzione **AND** opera bit per bit (**bitwise**); cio' significa che ciascun bit di **DEST**, viene sottoposto ad un **AND** con il bit corrispondente (stessa posizione) di **SRC**. Affinche' sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **AND**, viene memorizzato nell'operando **DEST**.

Riprendendo cio' che e' stato esposto nel Capitolo 5, consideriamo due **proposizioni logiche**, indicate simbolicamente con **A** e **B**; associamo, inoltre, il simbolo **1** alla proposizione logica vera (**TRUE**), e il simbolo **0** alla proposizione logica falsa (**FALSE**).
La composizione delle due proposizioni logiche attraverso l'operatore **AND** si scrive:
`A AND B`
e **risulta vera se, e solo se, entrambe le proposizioni sono vere**.
Poniamo, ad esempio:

`A = La capitale della Francia e' Parigi`
e:
`B = La capitale dell'Italia e' Berlino`

In questo caso otteniamo:
`A AND B = 0`
Infatti, la proposizione **A** e' vera, mentre la proposizione **B** e' falsa; in sostanza, alla precedente relazione dobbiamo attribuire il significato di:
`A e B`

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verita' (**truth table**) relativa all'operatore **AND** logico; nella parte sinistra della Figura 1 viene mostrato anche il simbolo logico di questo operatore.

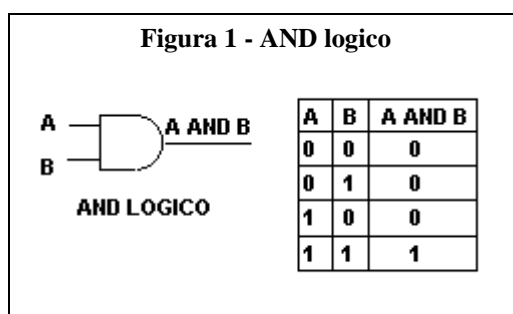



Figura 1 - AND logico

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Come esempio pratico, poniamo **BX=1000111010100110b**, **CX=0110100111110010b**, ed eseguiamo la seguente istruzione:
`and bx, cx`
In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
            BX = 1000111010100110b
            CX = 0110100111110010b
----------------------------------
BX = BX AND CX = 0000100010100010b
```

Per pervenire a questo risultato, la **CPU** esegue le seguenti elaborazioni (a partire dai bit meno significativi dei due operandi):

`0 AND 0 = 0` (**AND** tra i bit in posizione **0**)

`1 AND 1 = 1` (**AND** tra i bit in posizione **1**)

`1 AND 0 = 0` (**AND** tra i bit in posizione **2**)

e cosi' via.

19.1.1 Istruzione AND con maschere di bit

L'istruzione **AND** puo' essere utilizzata per portare a livello logico **0**, determinati bit di un numero binario; osserviamo, infatti, che indipendentemente dal valore (**0** o **1**) di una proposizione logica **A**, risulta:
`A AND 0 = 0` e `A AND 1 = A`
In sostanza, un **AND** tra **A** e **0** produce, in ogni caso, **A=0**; invece, un **AND** tra **A** e **1** lascia inalterato il contenuto di **A**.
In gergo tecnico, l'azione che consiste nel portare a livello logico **0** un bit di un numero binario, si definisce **clearing** (lucidare, rendere trasparente).

Come esempio pratico poniamo **AX=1001110010000011b**, e supponiamo di voler portare a livello logico **0**, i bit di **AX** in posizione **1**, **7**, **11** e **15**; prima di tutto, definiamo la seguente costante simbolica:
`BITMASK_AND = 0111011101111101b`
Come si puo' notare, gli unici bit di **BITMASK_AND** che valgono **0**, sono proprio quelli in posizione **1**, **7**, **11** e **15**; a questo punto, possiamo scrivere l'istruzione:
`and ax, BITMASK_AND`

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
                   AX = 1001110010000011b
          BITMASK_AND = 0111011101111101b
-------------------------------------------
AX = AX AND BITMASK_AND = 0001010000000001b
```

Come possiamo notare, i bit di **AX** in posizione **1**, **7**, **11** e **15** sono stati portati a **0**, mentre tutti gli altri sono rimasti inalterati; una costante simbolica come **BITMASK_AND**, viene definita **bit mask** (maschera di bit).

19.1.2 AND logico tra operandi di ampiezza arbitraria

L'istruzione **AND** opera bit per bit, per cui puo' essere applicata con estrema semplicita', anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.
Se, ad esempio, abbiamo a che fare con operandi a **128** bit, possiamo suddividere ciascuno di essi in **4 DWORD**; a questo punto, dobbiamo effettuare un **AND** tra ciascuna coppia di **DWORD** corrispondenti.

Consideriamo le seguenti definizioni:

```
Num1     dd    3CF218A6h, 2B96CDEAh, 6F9C482Dh, 881E63CAh
Num2     dd    48FB1DC9h, 66D41695h, 3C9DF1AFh, 7BB3F28Dh
Num3     dd    4 dup (0)
```

Se vogliamo ottenere:
`Num3 = Num1 AND Num2`
non dobbiamo fare altro che calcolare:

```
Num3[0] = Num1[0] AND Num2[0] = 3CF218A6h AND 48FB1DC9h = 08F21880h

Num3[4] = Num1[4] AND Num2[4] = 2B96CDEAh AND 66D41695h = 22940480h
```

```
Num3[8] = Num1[8] AND Num2[8] = 6F9C482Dh AND 3C9DF1AFh = 2C9C402Dh

Num3[12] = Num1[12] AND Num2[12] = 881E63CAh AND 7BB3F28Dh = 08126288h
```

Ovviamente, questo procedimento non e' applicabile ai numeri **BCD** (e cio' vale per tutte le altre istruzioni logiche illustrate in questo capitolo); in una eventualita' del genere, dobbiamo prima convertire i numeri **BCD** in binario!

19.1.3 Effetti provocati da AND sugli operandi e sui flags

L'esecuzione dell'istruzione **AND** tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione ad istruzioni del tipo:

```
and si, [si]
```

Dopo l'esecuzione di questa istruzione, la componente **Offset** contenuta in **SI**, viene sovrascritta dal risultato prodotto da **AND**!

La possibilita' di effettuare un **AND** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:

```
and dx, dx
```

Come si puo' facilmente constatare, l'esecuzione di una tale istruzione non provoca nessuna modifica del contenuto di **DX**; infatti:
`0 AND 0 = 0` e `1 AND 1 = 1`
Sfruttando questo aspetto, possiamo utilizzare **AND** per sapere se il contenuto di un registro vale zero; supponiamo, ad esempio, di avere **DX=0111001101001101b**. Eseguendo ora l'istruzione:

```
and dx, dx
```

otteniamo:

```
            DX = 0111001101001101b
            DX = 0111001101001101b
----------------------------------
DX = DX AND DX = 0111001101001101b
```

In sostanza, il risultato della precedente istruzione e' zero se, e solo se, tutti i bit di **DX** valgono **0**; se almeno un bit di **DX** vale **1**, il risultato della precedente istruzione e' diverso da zero. Come viene spiegato piu' avanti, per sapere se abbiamo ottenuto un risultato nullo, ci basta consultare **ZF**; e' importante anche ribadire che la precedente istruzione, non altera il contenuto di **DX**!

L'esecuzione dell'istruzione **AND**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **AND**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.2 L'istruzione TEST

Con il mnemonico **TEST** si indica l'istruzione **logical compare** (comparazione logica mediante **AND**); lo scopo di questa istruzione e' quello di effettuare una comparazione di tipo **AND** logico, tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
TEST  Reg, Reg
TEST  Reg, Mem
TEST  Mem, Reg
TEST  Reg, Imm
TEST  Mem, Imm
```

L'istruzione **TEST** opera bit per bit (**bitwise**); cio' significa che ciascun bit di **DEST**, viene sottoposto ad un **AND** con il bit corrispondente (stessa posizione) di **SRC**. Affinche' sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **TEST**, viene scartato.

In sostanza, l'istruzione **TEST** e' del tutto simile a **AND**; la differenza fondamentale sta nel fatto che il risultato prodotto da **TEST**, viene scartato. Il contenuto dell'operando **DEST** non subisce quindi nessuna modifica; in effetti, in analogia a **CMP** (comparazione aritmetica), l'istruzione **TEST** (comparazione logica) e' stata concepita proprio per permettere di "testare" lo stato di determinati bit dell'operando **DEST**, senza alterarne il contenuto.
Ad eccezione di questo aspetto, tutto cio' che e' stato esposto in relazione all'istruzione **AND** si applica quindi anche all'istruzione **TEST**.

19.2.1 Istruzione TEST con maschere di bit

Come e' stato appena spiegato, l'istruzione **TEST** e' stata concepita per permettere di "testare" lo stato di determinati bit dell'operando **DEST**, senza alterarne il contenuto; vediamo alcuni esempi pratici.

Poniamo **BH=01101101b**, e supponiamo di voler sapere se il bit in posizione **3** di **BH** vale **1**; a tale proposito, possiamo servirci di una maschera di bit attraverso l'istruzione:
```
test bh, 00001000b
```
Come si puo' notare, solo il bit in posizione **3** della maschera di bit, vale **1**; in base a quanto e' stato esposto per l'istruzione **AND**, possiamo affermare che la precedente istruzione produce un risultato diverso da zero se, e solo se, il bit in posizione **3** di **BH** vale **1**. Se, e solo se, il bit in posizione **3** di **BH** vale **0**, otteniamo un risultato nullo; come viene chiarito piu' avanti, per conoscere il risultato prodotto dalla precedente istruzione, possiamo servirci del flag **ZF**.

Dalle considerazioni appena esposte, si intuisce che possiamo utilizzare l'istruzione **TEST**, anche per sapere se un numero intero con segno in complemento a **2**, e' positivo o negativo; a tale proposito, dobbiamo osservare innanzi tutto che il bit di segno e' quello piu' significativo.
Per sapere allora se il registro **AL** contiene un numero positivo o negativo, ci basta scrivere l'istruzione:
```
test al, 10000000b
```
o, in modo equivalente:
```
test al, 80h
```
Questa istruzione fornisce un risultato nullo se, e solo se, il bit piu' significativo di **AL** vale **0** (numero positivo); se, e solo se, il bit piu' significativo di **AL** vale **1** (numero negativo), otteniamo

un risultato diverso da zero.
Come viene chiarito piu' avanti, per conoscere il risultato prodotto dalla precedente istruzione, possiamo servirci del flag **ZF** o di **SF**.

Osserviamo che, se nei precedenti due esempi avessimo utilizzato l'istruzione **AND**, avremmo alterato il contenuto di **DEST**; utilizzando invece **TEST**, possiamo svolgere questo tipo di verifiche in modo assolutamente sicuro.

19.2.2 TEST tra operandi di ampiezza arbitraria

L'istruzione **TEST** opera bit per bit, per cui puo' essere applicata con estrema semplicita', anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.

Supponiamo, ad esempio, di voler sapere se il contenuto a **96** bit di un dato chiamato **BigNum1**, rappresenta un numero positivo o negativo (in complemento a **2**); come al solito, dobbiamo partire dal presupposto che il bit di segno e' sempre quello piu' significativo.
In base al fatto che **BigNum1** e' formato da **96** bit, cioe' da **12** byte, possiamo scrivere la semplice istruzione:

```
test byte ptr BigNum1[11], 80h
```

19.2.3 Effetti provocati da TEST sugli operandi e sui flags

L'esecuzione dell'istruzione **TEST** tra **SRC** e **DEST**, preserva il contenuto di entrambi gli operandi; infatti, il risultato del test viene scartato.

La possibilita' di effettuare un **TEST** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:

```
test dx, dx
```

In questo caso, valgono tutte le considerazioni gia' esposte per l'istruzione **AND**.

L'esecuzione dell'istruzione **TEST**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **TEST**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.3 L'istruzione <u>NOT</u>

Con il mnemonico **NOT** si indica l'istruzione **one's complement negation** (negazione in complemento a uno); lo scopo di questa istruzione e' quello di invertire lo stato di tutti i bit dell'unico operando **DEST**, specificato esplicitamente dal programmatore.
Non essendo possibile modificare il contenuto di un **Imm**, le uniche forme lecite per l'istruzione **NOT** sono le seguenti:

```
NOT   Reg
NOT   Mem
```

L'istruzione **NOT** opera bit per bit (**bitwise**); cio' significa che ciascun bit di **DEST**, viene sottoposto ad una inversione (se vale **0** diventa **1** e viceversa). Il risultato prodotto da **NOT**, viene memorizzato nello stesso operando **DEST**.

Il complemento a uno di una proposizione logica **A** si scrive:
`NOT A`
e **risulta vera se, e solo se, la proposizione e' falsa**.

Poniamo, ad esempio:

`A = La capitale della Francia e' Parigi`

In questo caso otteniamo:
`NOT A = 0`
Infatti, stiamo negando una proposizione vera; stiamo affermando cioe', che la capitale della Francia **NON** e' Parigi.

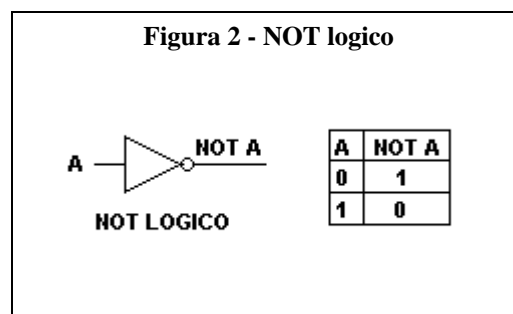
Poniamo, ad esempio:

`A = La capitale dell'Italia e' Berlino`

In questo caso otteniamo:
`NOT A = 1`
Infatti, stiamo negando una proposizione falsa; stiamo affermando cioe', che la capitale dell'Italia **NON** e' Berlino.
In sostanza, alle precedenti relazioni dobbiamo assegnare il significato di:
`negazione di A`

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verita' (**truth table**) relativa all'operatore **NOT** logico; nella parte sinistra della Figura 2 viene mostrato anche il simbolo logico di questo operatore.

Figura 2 - NOT logico

A	NOT A
0	1
1	0

Come esempio pratico, poniamo **CX=1000111010100110b**, ed eseguiamo la seguente istruzione:
`not cx`
In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
         CX = 1000111010100110b
----------------------------------
CX = NOT CX = 0111000101011001b
```

Per pervenire a questo risultato, la **CPU** esegue le seguenti elaborazioni (a partire dal bit meno significativo dell'operando):

`NOT 0 = 1` (**NOT** del bit in posizione **0**)

`NOT 1 = 0` (**NOT** del bit in posizione **1**)

`NOT 1 = 0` (**NOT** del bit in posizione **2**)

e cosi' via.

19.3.1 Complemento a 1 di un operando di ampiezza arbitraria

L'istruzione **NOT** opera bit per bit, per cui puo' essere applicata con estrema semplicita', anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere l'operando, in tante parti direttamente gestibili dalla **CPU**.
Consideriamo, ad esempio, la seguente definizione:

```
BigNum1 dd 3CF28615h, 8FB489DDh, 1C8DE4FFh
```

Se vogliamo ottenere il complemento a uno di **BigNum1**, non dobbiamo fare altro che calcolare:

```
BigNum1[0] = NOT BigNum1[0] = NOT 3CF28615h = C30D79EAh

BigNum1[4] = NOT BigNum1[4] = NOT 8FB489DDh = 704B7622h

BigNum1[8] = NOT BigNum1[8] = NOT 1C8DE4FFh = E3721B00h
```

19.3.2 Complemento a 2 di un operando di ampiezza arbitraria

Supponiamo di aver definito un dato **Num1** il cui contenuto rappresenta un numero intero con segno in complemento a **2**; a questo punto, vogliamo cambiare di segno lo stesso contenuto di **Num1**.
Come sappiamo, se l'ampiezza in bit di **Num1** e' direttamente gestibile dalla **CPU**, possiamo servirci semplicemente della sola istruzione **NEG**; se, invece, l'ampiezza in bit di **Num1** eccede la capacita' massima dei registri della **CPU**, possiamo sfruttare il fatto che, come e' stato gia' dimostrato nei precedenti capitoli:

```
NEG(Num1) = NOT(Num1) + 1
```

Come esempio pratico, supponiamo di operare sui numeri interi con segno a **64** bit; in tal caso, possiamo rappresentare tutti i numeri interi con segno compresi tra il limite minimo:
$-(2^{64} / 2)$
e il limite massimo:
$+((2^{64} / 2) - 1)$
Come al solito, il bit di segno e' quello piu' significativo (che in questo caso e' il bit in posizione **63**); tutti i numeri il cui bit piu' significativo vale **0** sono positivi, mentre tutti i numeri il cui bit piu' significativo vale **1** sono negativi.

Consideriamo ora il numero **-3**, che a **64** bit in complemento a **2** si scrive:
2^{64} `- 3 = 18446744073709551616 - 3 = 18446744073709551613 = FFFFFFFFFFFFFFFDh`
Negando ora questo numero, dovremmo ottenere **+3**, cioe' **0000000000000003h**; in base a quanto e' stato detto in precedenza, per negare un numero intero con segno a **64** bit, dobbiamo prima sottoporlo ad un **NOT**, sommando poi **1** al risultato ottenuto.

Partiamo quindi con la seguente definizione:

```
Num1 dq -3 ; FFFFFFFFFFFFFFFDh
```

Il seguente codice effettua il cambiamento di segno di **-3**:

```
; complemento a 1 di Num1

not      dword ptr Num1[0]    ; Num1[0] = NOT FFFFFFFDh = 00000002h
not      dword ptr Num1[4]    ; Num1[4] = NOT FFFFFFFFh = 00000000h

; incremento di 1 del risultato (complemento a 2 di Num1)

add      dword ptr Num1[0], 1 ; Num1[0] = 00000002h + 1h = 00000003h
adc      dword ptr Num1[4], 0 ; Num1[4] = 00000000h + 0h = 00000000h
```

E' importante ricordare che in questo caso, per incrementare di **1** il contenuto di **Num1[0]**, non possiamo utilizzare **INC**; infatti, l'istruzione **INC** non modifica il flag **CF** di cui ha bisogno la successiva istruzione **ADC**!

Se vogliamo visualizzare il contenuto esadecimale di **Num1**, possiamo procedere nel solito modo con l'ausilio di **writeHex32**; in questo caso, l'output deve iniziare da **Num1[4]**.

19.3.3 Effetti provocati da NOT sugli operandi e sui flags

L'esecuzione dell'istruzione **NOT** modifica il contenuto del suo unico operando **DEST**; tale operando viene sovrascritto dal risultato prodotto dall'istruzione stessa.

L'esecuzione dell'istruzione **NOT**, non modifica nessun campo del **Flags Register**.

19.4 L'istruzione OR

Con il mnemonico **OR** si indica l'istruzione **logical inclusive OR** (**OR** logico inclusivo); lo scopo di questa istruzione e' quello di effettuare un **OR** logico inclusivo tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
OR    Reg, Reg
OR    Reg, Mem
OR    Mem, Reg
OR    Reg, Imm
OR    Mem, Imm
```

L'istruzione **OR** opera bit per bit (**bitwise**); cio' significa che ciascun bit di **DEST**, viene sottoposto ad un **OR** con il bit corrispondente (stessa posizione) di **SRC**. Affinche' sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **OR**, viene memorizzato nell'operando **DEST**.

La composizione di due proposizioni logiche **A** e **B** attraverso l'operatore **OR** si scrive:
`A OR B`
e **risulta vera se, e solo se, almeno una di esse e' vera**.

Poniamo, ad esempio:

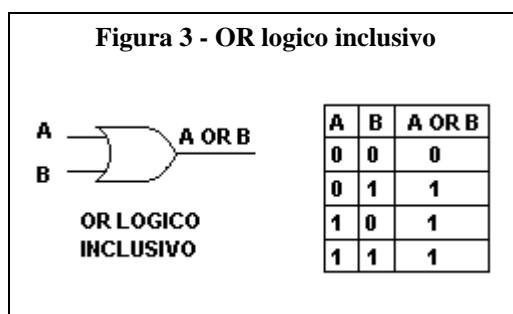
`A = La capitale della Francia e' Parigi`
e:
`B = La capitale dell'Italia e' Berlino`

In questo caso otteniamo:

```
A OR B = 1
```

Infatti, almeno una delle due proposizioni (la **A**) e' vera; in sostanza, all'operatore **OR** dobbiamo attribuire il significato di:

```
A o B o entrambe
```

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verita' (**truth table**) relativa all'operatore **OR** logico inclusivo; nella parte sinistra della Figura 3 viene mostrato anche il simbolo logico di questo operatore.

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Come esempio pratico, poniamo **BX=1000111010100110b**, **CX=0110100111110010b**, ed eseguiamo la seguente istruzione:

```
or bx, cx
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
           BX = 1000111010100110b
           CX = 0110100111110010b
---------------------------------
BX = BX OR CX = 1110111111110110b
```

Per pervenire a questo risultato, la **CPU** esegue le seguenti elaborazioni (a partire dai bit meno significativi dei due operandi):

`0 OR 0 = 0` (**OR** tra i bit in posizione **0**)

`1 OR 1 = 1` (**OR** tra i bit in posizione **1**)

`1 OR 0 = 1` (**OR** tra i bit in posizione **2**)

e cosi' via.

19.4.1 Istruzione OR con maschere di bit

L'istruzione **OR** puo' essere utilizzata per portare a livello logico **1**, determinati bit di un numero binario; osserviamo, infatti, che indipendentemente dal valore (**0** o **1**) di una proposizione logica **A**, risulta:
`A OR 0 = A` e `A OR 1 = 1`
In sostanza, un **OR** tra **A** e **0** lascia inalterato il contenuto di **A**; invece, un **OR** tra **A** e **1** produce, in ogni caso, **A=1**.
In gergo tecnico, l'azione che consiste nel portare a livello logico **1** un bit di un numero binario, si definisce **setting** (settaggio).
Come esempio pratico poniamo **AX=1001110010000011b**, e supponiamo di voler portare a livello logico **1**, i bit di **AX** in posizione **3**, **4**, **5** e **8**; prima di tutto, definiamo la seguente costante simbolica:
`BITMASK_OR = 0000000100111000b`
Come si puo' notare, gli unici bit di **BITMASK_OR** che valgono **1**, sono proprio quelli in posizione **3**, **4**, **5** e **8**; a questo punto, possiamo scrivere l'istruzione:
`or ax, BITMASK_OR`
In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
                   AX = 1001110010000011b
           BITMASK_OR = 0000000100111000b
-----------------------------------------
AX = AX OR BITMASK_OR = 1001110110111011b
```

Come possiamo notare, i bit di **AX** in posizione **3**, **4**, **5** e **8** sono stati portati a **1**, mentre tutti gli altri sono rimasti inalterati.

19.4.2 OR logico inclusivo tra operandi di ampiezza arbitraria

L'istruzione **OR** opera bit per bit, per cui puo' essere applicata con estrema semplicita', anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.
Se, ad esempio, abbiamo a che fare con operandi a **128** bit, possiamo suddividere ciascuno di essi in **4 DWORD**; a questo punto, dobbiamo effettuare un **OR** tra ciascuna coppia di **DWORD** corrispondenti.

Consideriamo le seguenti definizioni:

```
Num1     dd    3CF218A6h, 2B96CDEAh, 6F9C482Dh, 881E63CAh
Num2     dd    48FB1DC9h, 66D41695h, 3C9DF1AFh, 7BB3F28Dh
Num3     dd    4 dup (0)
```

Se vogliamo ottenere:
`Num3 = Num1 OR Num2`
non dobbiamo fare altro che calcolare:

```
Num3[0] = Num1[0] OR Num2[0] = 3CF218A6h OR 48FB1DC9h = 7CFB1DEFh

Num3[4] = Num1[4] OR Num2[4] = 2B96CDEAh OR 66D41695h = 6FD6DFFFh

Num3[8] = Num1[8] OR Num2[8] = 6F9C482Dh OR 3C9DF1AFh = 7F9DF9AFh

Num3[12] = Num1[12] OR Num2[12] = 881E63CAh OR 7BB3F28Dh = FBBFF3CFh
```

19.4.3 Effetti provocati da OR sugli operandi e sui flags

L'esecuzione dell'istruzione **OR** tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione ad istruzioni del tipo:
```
or bx, cs:[bx+si+03h]
```
Dopo l'esecuzione di questa istruzione, la componente **Offset** contenuta in **BX**, viene sovrascritta dal risultato prodotto da **OR**!

La possibilita' di effettuare un **OR** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:
```
or dx, dx
```
Come si puo' facilmente constatare, l'esecuzione di una tale istruzione non provoca nessuna modifica del contenuto di **DX**; infatti:
`0 OR 0 = 0` e `1 OR 1 = 1`
Sfruttando questo aspetto, possiamo utilizzare **OR** per sapere se il contenuto di un registro vale zero; in tal caso, si applicano tutte le considerazioni gia' svolte per l'istruzione **AND**.

L'esecuzione dell'istruzione **OR**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **OR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.5 L'istruzione XOR

Con il mnemonico **XOR** si indica l'istruzione **logical eXclusive OR** (**OR** logico esclusivo); lo scopo di questa istruzione e' quello di effettuare un **OR** logico esclusivo tra i contenuti dei due operandi, **SRC** e **DEST**, specificati esplicitamente dal programmatore.
Sono permesse tutte le combinazioni tra **SRC** e **DEST**, ad eccezione di quelle illegali o prive di senso; possiamo avere quindi i seguenti casi:

```
XOR   Reg, Reg
XOR   Reg, Mem
XOR   Mem, Reg
XOR   Reg, Imm
XOR   Mem, Imm
```

L'istruzione **XOR** opera bit per bit (**bitwise**); cio' significa che ciascun bit di **DEST**, viene sottoposto ad uno **XOR** con il bit corrispondente (stessa posizione) di **SRC**. Affinche' sia possibile un confronto di tipo **bitwise**, i due operandi devono avere la stessa ampiezza in bit; il risultato prodotto da **XOR**, viene memorizzato nell'operando **DEST**.

La composizione di due proposizioni logiche **A** e **B** attraverso l'operatore **XOR** si scrive:
```
A XOR B
```
e **risulta vera se, e solo se, una di esse e' vera e l'altra e' falsa**.

Poniamo, ad esempio:

```
A = La capitale della Francia e' Parigi
```
e:
```
B = La capitale dell'Italia e' Roma
```

In questo caso otteniamo:
`A XOR B = 0`
Infatti, entrambe le proposizioni sono vere; in sostanza, all'operatore **XOR** dobbiamo attribuire il significato di:
`(A e NOT(B)) o (NOT(A) e B)`

In base alle considerazioni appena esposte, possiamo ricavare la tabella della verita' (**truth table**) relativa all'operatore **OR** logico esclusivo; nella parte centrale della Figura 4 viene mostrato anche il simbolo logico di questo operatore, mentre nella parte sinistra vediamo una possibile realizzazione dello **XOR** attraverso porte **AND**, **OR** e **NOT** (la porta **NOT** viene rappresentata mediante un cerchietto).

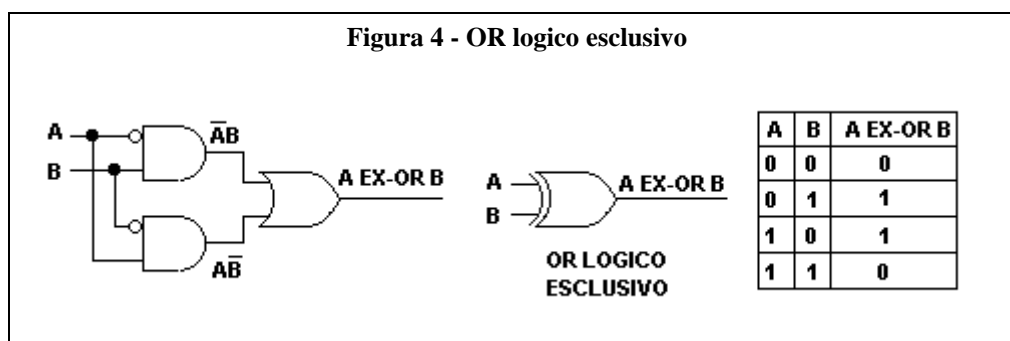



Figura 4 - OR logico esclusivo

A	B	A EX-OR B
0	0	0
0	1	1
1	0	1
1	1	0

Come esempio pratico, poniamo **BX=1000111010100110b**, **CX=0110100111110010b**, ed eseguiamo la seguente istruzione:
```
xor bx, cx
```

In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
            BX = 1000111010100110b
            CX = 0110100111110010b
---------------------------------
BX = BX XOR CX = 1110011101010100b
```

Per pervenire a questo risultato, la **CPU** esegue le seguenti elaborazioni (a partire dai bit meno significativi dei due operandi):

`0 XOR 0 = 0` (**XOR** tra i bit in posizione **0**)

`1 XOR 1 = 0` (**XOR** tra i bit in posizione **1**)

`1 XOR 0 = 1` (**XOR** tra i bit in posizione **2**)

e cosi' via.

19.5.1 OR logico esclusivo tra operandi di ampiezza arbitraria

L'istruzione **XOR** opera bit per bit, per cui puo' essere applicata con estrema semplicita', anche ad operandi di ampiezza arbitraria; in sostanza, non dobbiamo fare altro che suddividere ciascun operando, in tante parti direttamente gestibili dalla **CPU**.
Se, ad esempio, abbiamo a che fare con operandi a **96** bit, possiamo suddividere ciascuno di essi in **3 DWORD**; a questo punto, dobbiamo effettuare uno **XOR** tra ciascuna coppia di **DWORD** corrispondenti.

Consideriamo le seguenti definizioni:

```
Num1     dd    2B96CDEAh, 6F9C482Dh, 881E63CAh
Num2     dd    66D41695h, 3C9DF1AFh, 7BB3F28Dh
Num3     dd    3 dup (0)
```

Se vogliamo ottenere:
`Num3 = Num1 XOR Num2`
non dobbiamo fare altro che calcolare:

```
Num3[0] = Num1[0] XOR Num2[0] = 2B96CDEAh XOR 66D41695h = 4D42DB7Fh

Num3[4] = Num1[4] XOR Num2[4] = 6F9C482Dh XOR 3C9DF1AFh = 5301B982h

Num3[8] = Num1[8] XOR Num2[8] = 881E63CAh XOR 7BB3F28Dh = F3AD9147h
```

19.5.2 Effetti provocati da XOR sugli operandi e sui flags

L'esecuzione dell'istruzione **XOR** tra **SRC** e **DEST**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione ad istruzioni del tipo:
`xor di, ss:[di+05h]`
Dopo l'esecuzione di questa istruzione, la componente **Offset** contenuta in **DI**, viene sovrascritta dal risultato prodotto da **XOR**!

La possibilita' di effettuare uno **XOR** tra **Reg** e **Reg**, ci permette di scrivere anche istruzioni del tipo:
`xor dx, dx`
Come si puo' facilmente constatare, l'esecuzione di una tale istruzione provoca l'azzeramento del contenuto di **DX**; infatti:
`0 XOR 0 = 0` e `1 XOR 1 = 0`
Poniamo, ad esempio, **DX=1001110010000011b**, ed eseguiamo la seguente istruzione:
`xor dx, dx`
In presenza di questa istruzione, la **CPU** ottiene il seguente risultato:

```
            DX = 1001110010000011b
            DX = 1001110010000011b
----------------------------------
DX = DX XOR DX = 0000000000000000b
```

Il ricorso a **XOR** per l'azzeramento di un registro, e' una pratica molto diffusa tra i programmatori **Assembly**; si tenga presente, comunque, che non si ottiene nessun guadagno in velocita' rispetto alle analoghe istruzioni:

```
mov dx, 0
```

e:

```
sub dx, dx
```

L'esecuzione dell'istruzione **XOR**, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
La **CPU** pone, in ogni caso, **CF=0** e **OF=0**, mentre i campi **PF**, **ZF** e **SF** forniscono informazioni ordinarie relative al risultato prodotto da **XOR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.6 L'istruzione SHL

Con il mnemonico **SHL** si indica l'istruzione **SHift logical Left** (scorrimento logico verso sinistra); lo scopo di questa istruzione e' quello di effettuare uno scorrimento verso sinistra, dei bit dell'operando **DEST**, il cui contenuto viene trattato come numero intero senza segno. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a destra nell'operando **DEST**, vengono riempiti con degli zeri.
Per ogni singolo scorrimento, il bit piu' significativo dell'operando **DEST** trabocca da sinistra; il valore di tale bit, viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SHL**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHL** sono le seguenti:

```
SHL   Reg, 1
SHL   Reg, CL
SHL   Reg, Imm8
SHL   Mem, 1
SHL   Mem, CL
SHL   Mem, Imm8
```

Il codice macchina generale per l'istruzione **SHL**, e' formato dal campo **Opcode 110100vw** e dal campo **mod_100_r/m**; come si puo' notare, nel campo **Opcode** e' presente un bit indicato con **v**. Se **v=0**, allora l'operando **SRC** e' rappresentato dal valore immediato **1** (un solo scorrimento); se **v=1**, allora l'operando **SRC** e' rappresentato dall'half register **CL** (che contiene il numero di scorrimenti da effettuare). Con le **CPU 80286** e superiori, il numero di scorrimenti puo' essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina e' formato dal campo **Opcode 1100000w** e dal campo **mod_100_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
shl ax, cl
```

In presenza di questa istruzione, la **CPU** fa' scorrere di **5** posti verso sinistra, tutti i bit di **AX**; i **5** posti che si liberano a destra, vengono riempiti con degli zeri. La successione dei singoli scorrimenti produce quindi i seguenti risultati:

```
AX = 1110010110111000b, CF = 0

AX = 1100101101110000b, CF = 1

AX = 1001011011100000b, CF = 1

AX = 0010110111000000b, CF = 1

AX = 0101101110000000b, CF = 0
```

I singoli bit che traboccano da sinistra, vengono salvati nel flag **CF**; nel nostro caso, dopo **5** scorrimenti, l'ultimo bit traboccato da sinistra e' uno **0**, per cui alla fine otteniamo **AX=0101101110000000b** e **CF=0**.

Con le **CPU 8086**, il registro **CL** puo' contenere un qualsiasi valore compreso tra **0** e **255**; si tratta chiaramente di una situazione priva di senso, che in certi casi puo' comportare, per la **CPU**, tempi di elaborazione elevatissimi. Per rendercene conto, poniamo, ad esempio, **AX=0111010100101111b**, **CL=255**, ed eseguiamo la seguente istruzione:

```
shl ax, cl
```

Come si puo' facilmente constatare, dopo i primi **16** scorrimenti, tutti i **16** bit di **AX** sono traboccati da sinistra e sono stati sostituiti con **16** zeri; in sostanza, dopo i primi **16** scorrimenti si ottiene **AX=0**, per cui e' perfettamente inutile continuare con gli altri **239** scorrimenti.
L'aspetto piu' grave e' legato al fatto che le **CPU 8086**, implementano istruzioni logiche scarsamente ottimizzate; tali istruzioni, vengono quindi eseguite con una certa lentezza. Come si nota dalle tabelle, l'istruzione **SHL** con contatore **CL** viene eseguita dall'**8086** in **8** cicli di clock, ai quali bisogna aggiungere altri **4** cicli di clock per ogni scorrimento da **1** bit; se **CL** contiene il valore **255**, l'**8086** esegue l'istruzione in ben:

```
8 + (4 x 255) = 1028 cicli di clock
```

La frequenza di clock dell'**8086** e' di circa **5** milioni di cicli al secondo (**5 MHz**), per cui l'esecuzione dell'istruzione richiede un tempo abissale di:

```
1028 / 5000000 = 0.0002056 secondi = 0.2056 millisecondi
```

Questo problema e' stato eliminato a partire dalle **CPU 80286**; tali **CPU**, infatti, sottopongono il contenuto del contatore (**CL** o **Imm8**), ad un **AND** con il valore **00011111b**. Come sappiamo, una istruzione del tipo:

```
and cl, 00011111b
```

azzera i tre bit piu' significativi di **CL**, e lascia inalterati i restanti **5** bit. In sostanza, le **CPU 80286** e superiori, prendono in considerazione solamente i primi **5** bit del contatore; con **5** bit possiamo rappresentare un qualunque valore compreso tra **0** e **31**, piu' che sufficiente per operare con registri a **8**, **16** o **32** bit.
A tutto cio' bisogna aggiungere che le **CPU 80286** e superiori, implementano istruzioni logiche altamente ottimizzate; quando si utilizza **CL** o un **Imm8** come contatore, la **CPU 80286** richiede un solo ciclo di clock aggiuntivo per ogni bit di scorrimento. Nel caso delle **CPU 80386** e superiori, il numero di cicli di clock diventa addirittura indipendente dal numero di scorrimenti da effettuare; come si nota dalle tabelle, una **CPU 80486** a **33** MHz, esegue l'istruzione **SHL** in appena **2/3** cicli di clock, indipendentemente dal numero di scorrimenti specificati da **CL** o da un **Imm8**.
Nota importante.
Le considerazioni appena esposte sulla gestione del contatore **CL** o **Imm8** da parte delle **CPU 80286** e superiori, si applicano anche alle altre istruzioni di scorrimento e di rotazione dei bit,

illustrate nel seguito del capitolo; tali istruzioni sono rappresentate dai mnemonici **SAL**, **SHR**, **SAR**, **ROL**, **RCR**, **ROR**, **RCR**.

19.6.1 Significato matematico dell'istruzione SHL

Come abbiamo visto nel capitolo dedicato alla matematica del computer, moltiplicare un numero intero senza segno **m**, espresso in base **b**, per un fattore $\mathbf{b^n}$ (**n** intero positivo), equivale ad aggiungere **n** zeri alla destra dello stesso **m**; in questo modo, tutte le cifre di **m** vengono fatte scorrere di **n** posti verso sinistra.
Possiamo dire quindi che una istruzione del tipo:

```
shl ax, n
```

equivale a moltiplicare il contenuto binario di **AX** per $\mathbf{2^n}$.

Come verifica poniamo, **AX=0000101011110010b=2802**, ed eseguiamo l'istruzione:

```
shl ax, 4
```

In presenza di questa istruzione, la **CPU** sottopone il contenuto di **AX** ai seguenti **4** singoli scorrimenti:

```
AX = 0001010111100100b = 5604 = 2802 * 2, CF = 0

AX = 0010101111001000b = 11208 = 2802 * 4, CF = 0

AX = 0101011110010000b = 22416 = 2802 * 8, CF = 0

AX = 1010111100100000b = 44832 = 2802 * 16, CF = 0
```

Come si puo' notare, dopo **4** scorrimenti verso sinistra, il contenuto iniziale **2802** di **AX** risulta moltiplicato per $\mathbf{2^4=16}$!

Una **CPU 80486 DX** esegue la precedente istruzione in appena **2** cicli di clock; per ottenere lo stesso risultato, possiamo porre **AX=0000101011110010b=2802**, **BX=16**, ed eseguire l'istruzione:

```
mul bx
```

La stessa **CPU 80486 DX** esegue questa istruzione in **13** cicli di clock, cioe', oltre **6** volte piu' lentamente rispetto all'uso di **SHL**!
Proprio per questo motivo, ogni volta che si deve eseguire una moltiplicazione tra numeri interi senza segno, con uno dei fattori esprimibile nella forma $\mathbf{2^n}$ (**n** intero positivo), e' vivamente consigliabile l'utilizzo di **SHL** con contatore **n**.
Il programmatore che intende utilizzare **SHL** al posto di **MUL**, deve prestare pero' particolare attenzione al fatto che, i registri della **CPU** (e le locazioni di memoria), hanno una ampiezza in bit limitata; un numero eccessivo di scorrimenti verso sinistra, puo' portare quindi al trabocco di cifre significative dell'operando **DEST**, con un conseguente risultato privo di senso!
Per verificare questo aspetto, riprendiamo il precedente esempio nel quale, dopo **4** scorrimenti verso sinistra, avevamo ottenuto:

```
AX = 1010111100100000b = 44832 = 2802 * 16, CF = 0
```

Sottoponendo ora **AX** ad un ulteriore scorrimento di **1** bit verso sinistra, otteniamo:

```
AX = 0101111001000000b = 24128, CF = 1
```

Questo risultato e' sbagliato in quanto si e' verificato il trabocco da sinistra, di un bit di valore **1**; questa situazione viene segnalata da **CF=1**. In sostanza, il risultato corretto e' rappresentato dal

contenuto di **AX** e da un ulteriore bit (quello che e' traboccato da sinistra), per un totale di **17** bit; infatti, aggiungendo **1** alla sinistra del precedente risultato, si ottiene:

```
10101111001000000b = 89664 = 2802 * 2^5
```

In definitiva, una (pseudo) istruzione del tipo:
```
mul DEST, 2^n
```
e' sostituibile con una (pseudo) istruzione del tipo:
```
shl DEST, n
```
solo se siamo sicuri che il risultato finale e' interamente rappresentabile con l'ampiezza in bit di **DEST**.

19.6.2 Shift logical left su operandi di ampiezza arbitraria

Supponiamo di voler far scorrere verso sinistra, i bit di un numero intero senza segno di ampiezza arbitraria; in un caso del genere, la strada migliore da seguire consiste nel definire un procedimento generale, indipendente dall'ampiezza in bit dell'operando, e dal numero di scorrimenti che vogliamo effettuare.
Un tale procedimento ha bisogno di conoscere, l'indirizzo iniziale del dato da sottoporre a shifting, l'ampiezza in byte del dato stesso, e il numero di scorrimenti da effettuare; nel caso di una richiesta di scorrimento di **n** posti, possiamo pensare di ripetere per **n** volte, un procedimento che effettua un solo scorrimento alla volta.

Il punto fondamentale consiste allora nel trovare un metodo per effettuare uno scorrimento unitario verso sinistra; analizziamo una soluzione molto elementare, che ci permette anche di illustrare la potenza delle istruzioni logiche. Quando avremo a disposizione le procedure e le istruzioni per i loop, saremo in grado di ottimizzare tutti gli algoritmi presentati nel seguito del capitolo e nei capitoli precedenti.

Vogliamo far scorrere di un posto verso sinistra, tutti i bit del numero:
```
Num1 = 0011110011110010b
```
Poniamo **AH=0**, e carichiamo in **AL** il byte meno significativo di **Num1** (**AL=11110010b**); in seguito ad una istruzione:
```
shl ax, 1
```
otteniamo:
```
AX = 0000000111100100b
```
In sostanza, il bit piu' significativo di **AL** trabocca da sinistra, e finisce nel bit meno significativo di **AH**; si ha quindi **AH=00000001b** e **AL=11100100b**.
Salviamo il contenuto di **AL** in **Num1[0]**, e il contenuto di **AH**, ad esempio, in **BL**; ripetiamo ora il procedimento sul secondo byte di **Num1**. Poniamo quindi **AH=0** e **AL=00111100b**; in seguito ad una istruzione:
```
shl ax, 1
```
otteniamo:
```
AX = 0000000001111000b
```
In sostanza, il bit piu' significativo di **AL** trabocca da sinistra, e finisce nel bit meno significativo di **AH**; si ha quindi **AH=00000000b** e **AL=01111000b**.
Il bit che avevamo salvato in **BL** deve essere ora posizionato nel bit meno significativo di **AL**; a tale proposito, ci basta eseguire la seguente istruzione:
```
AL = AL OR BL = 01111000b OR 00000001b = 01111001b
```
Salviamo il contenuto di **AL** in **Num1[1]**; a questo punto possiamo notare che:
```
Num1 = 0111100111100100b
```
Il registro **AH** contiene il bit piu' significativo di **Num1**, traboccato da sinistra (nel nostro caso, **AH=00000000b**); appare anche evidente il fatto che, il procedimento appena descritto, puo' essere

applicato a numeri interi senza segno di qualunque ampiezza.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 00011111b ; 000111110011110010010010b
```

Il codice che esegue lo scorrimento unitario verso sinistra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 0 di bigNum

xor      ah, ah              ; ah = 0
mov      al, BigNum[0]       ; al = 10010010b
shl      ax, 1               ; ah = 00000001b, al = 00100100b
mov      BigNum[0], al       ; BigNum[0] = 00100100b
mov      bl, ah              ; bl = ah = 00000001b

; shifting unitario del BYTE n. 1 di bigNum

xor      ah, ah              ; ah = 0
mov      al, BigNum[1]       ; al = 00111100b
shl      ax, 1               ; ah = 00000000b, al = 01111000b
or       al, bl              ; al = 01111001b
mov      BigNum[1], al       ; BigNum[1] = 01111001b
mov      bl, ah              ; bl = ah = 00000000b

; shifting unitario del BYTE n. 2 di bigNum

xor      ah, ah              ; ah = 0
mov      al, BigNum[2]       ; al = 00011111b
shl      ax, 1               ; ah = 00000000b, al = 00111110b
or       al, bl              ; al = 00111110b
mov      BigNum[2], al       ; BigNum[2] = 00111110b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 001111100111100100100100b
```

Il bit piu' significativo di **BigNum**, traboccato da sinistra, si trova memorizzato nel bit meno significativo di **AH**; naturalmente, e' fondamentale ricordare che la locazione di memoria riservata a **BigNum**, deve avere spazio sufficiente per contenere il risultato dello shifting.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.6.3 Effetti provocati da SHL sugli operandi e sui flags

L'esecuzione dell'istruzione **SHL**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione ad un caso delicato, rappresentato da istruzioni del tipo:

```
shl cx, cl
```

(ovviamente, sarebbe opportuno evitare l'inserimento nei propri programmi, di questo genere di istruzioni).
Per capire bene il funzionamento della precedente istruzione, dobbiamo ricordare che le **CPU 80286** e superiori, effettuano un **AND** tra il contenuto di **CL** e la maschera di bit **00011111b**; vediamo allora quello che succede ponendo **CX=0011101011100110b** ed eseguendo l'istruzione:

```
shl cx, cl
```

Prima di tutto, la **CPU** calcola:

```
CL = CL AND 00011111b = 11100110b AND 00011111b = 00000110b = 6
```

Di conseguenza, il contenuto di **CX** viene alterato e diventa **0011101000000110b**; i bit di **CX** vengono fatti scorrere di **6** posti verso sinistra, con il risultato finale **CX=1000000110000000b** (**CF=0**).

L'esecuzione dell'istruzione **SHL** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **SHL** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **SHL** provoca la modifica del bit piu' significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla sinistra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SHL**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.7 L'istruzione SAL

Con il mnemonico **SAL** si indica l'istruzione **Shift Arithmetic Left** (scorrimento aritmetico verso sinistra); lo scopo di questa istruzione e' quello di effettuare uno scorrimento verso sinistra, dei bit dell'operando **DEST**, il cui contenuto viene trattato come numero intero con segno in complemento a **2**. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a destra nell'operando **DEST**, vengono riempiti con degli zeri.
Per ogni singolo scorrimento, il bit piu' significativo dell'operando **DEST** trabocca da sinistra; il valore di tale bit, viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SAL**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SAL** sono le seguenti:

```
SAL   Reg, 1
SAL   Reg, CL
SAL   Reg, Imm8
SAL   Mem, 1
SAL   Mem, CL
SAL   Mem, Imm8
```

A differenza di **SHL** che tratta il contenuto di **DEST** come un numero intero senza segno, l'istruzione **SAL** effettua il proprio lavoro trattando il contenuto di **DEST** come un numero intero con segno in complemento a **2**; siccome pero' il bit di segno di **DEST** e' quello piu' a sinistra, e gli scorrimenti si svolgono, appunto, verso sinistra, gli effetti prodotti da **SAL** sono assolutamente identici a quelli prodotti da **SHL**!
Le due istruzioni **SHL** e **SAL** sono del tutto equivalenti, e risultano quindi interscambiabili; proprio per questo motivo, il codice macchina di **SAL** e' identico a quello di **SHL**. L'istruzione **SAL** viene resa disponibile solo per motivi di simmetria; al ruolo svolto dalla coppia **SHL**, **SAL**, si

contrappone, infatti, quello della coppia **SHR**, **SAR**, illustrata piu' avanti.

19.7.1 Significato matematico dell'istruzione SAL

Il significato matematico dell'istruzione **SAL** e' del tutto simile a quello gia' illustrato per l'istruzione **SHL**; piu' precisamente, possiamo dire che una (pseudo) istruzione del tipo:

```
sal DEST, n
```

e' equivalente ad una (pseudo) istruzione del tipo:

```
imul DEST, 2^n
```

Non dobbiamo dimenticare pero' che questa volta, l'operando **DEST** rappresenta un numero intero con segno in complemento a **2**; come si puo' facilmente intuire, la situazione appare piu' delicata, a causa del fatto che il bit di segno dell'operando **DEST** e' quello piu' a sinistra.

Per analizzare questo aspetto poniamo, **AX=1101110011110001b=-8975**, ed eseguiamo l'istruzione:

```
sal ax, 1
```

Dopo l'esecuzione di questa istruzione, otteniamo:

```
AX = 1011100111100010b = -17950 = (-8975) * 2^1, CF = 1, OF = 0
```

Il risultato e' esatto in quanto **SAL** non ha provocato la perdita del bit di segno di **AX**; infatti, il bit piu' significativo di **AX** continua a valere **1**. La **CPU** segnala questa situazione ponendo **OF=0**, in quanto, moltiplicando un numero negativo per un numero positivo, abbiamo ottenuto, correttamente, un numero negativo.

Vediamo pero' quello che succede con una ulteriore istruzione:

```
sal ax, 1
```

Dopo l'esecuzione di questa istruzione, otteniamo:

```
AX = 0111001111000100b = +29636, CF = 1, OF = 1
```

Il risultato e' sbagliato in quanto **SAL** ha provocato la perdita del bit di segno di **AX**; infatti, il bit piu' significativo di **AX** e' passato da **1** a **0**. La **CPU** segnala questa situazione ponendo **OF=1**, in quanto, moltiplicando un numero negativo per un numero positivo, abbiamo ottenuto, erroneamente, un numero positivo.

In definitiva, una (pseudo) istruzione del tipo:

```
imul DEST, 2^n
```

e' sostituibile con una (pseudo) istruzione del tipo:

```
sal DEST, n
```

solo se siamo sicuri che il risultato finale e' interamente rappresentabile con l'ampiezza in bit di **DEST**.

19.7.2 Shift arithmetic left su operandi di ampiezza arbitraria

Come e' stato gia' sottolineato, gli effetti prodotti da **SAL** sull'operando **DEST**, sono assolutamente identici a quelli prodotti da **SHL**; di conseguenza, se vogliamo applicare **SAL** ad operandi di ampiezza arbitraria, dobbiamo procedere esattamente come nell'esempio mostrato per **SHL**.

19.7.3 Effetti provocati da SAL sugli operandi e sui flags

Anche in relazione agli effetti provocati da **SAL** sugli operandi e sui flags, valgono tutte le considerazioni gia' esposte per **SHL**.

19.8 L'istruzione SHR

Con il mnemonico **SHR** si indica l'istruzione **SHift logical Right** (scorrimento logico verso destra); lo scopo di questa istruzione e' quello di effettuare uno scorrimento verso destra, dei bit dell'operando **DEST**, il cui contenuto viene trattato come numero intero senza segno. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a sinistra nell'operando **DEST**, vengono riempiti con degli zeri.
Per ogni singolo scorrimento, il bit meno significativo dell'operando **DEST** trabocca da destra; il valore di tale bit, viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SHR**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHR** sono le seguenti:

```
SHR   Reg, 1
SHR   Reg, CL
SHR   Reg, Imm8
SHR   Mem, 1
SHR   Mem, CL
SHR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **SHR**, e' formato dal campo **Opcode 110100vw** e dal campo **mod_101_r/m**; se **v=0**, allora l'operando **SRC** e' rappresentato dal valore immediato **1** (un solo scorrimento), mentre se **v=1**, allora l'operando **SRC** e' rappresentato dall'half register **CL** (che contiene il numero di scorrimenti da effettuare). Con le **CPU 80286** e superiori, il numero di scorrimenti puo' essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina e' formato dal campo **Opcode 1100000w** e dal campo **mod_101_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:
```
shr ax, cl
```
In presenza di questa istruzione, la **CPU** fa' scorrere di **5** posti verso destra, tutti i bit di **AX**; i **5** posti che si liberano a sinistra, vengono riempiti con degli zeri. La successione dei singoli scorrimenti produce quindi i seguenti risultati:

```
AX = 0011100101101110b, CF = 0

AX = 0001110010110111b, CF = 0

AX = 0000111001011011b, CF = 1

AX = 0000011100101101b, CF = 1

AX = 0000001110010110b, CF = 1
```

I singoli bit che traboccano da destra, vengono salvati nel flag **CF**; nel nostro caso, dopo **5** scorrimenti, l'ultimo bit traboccato da destra e' un **1**, per cui alla fine otteniamo **AX=0000001110010110b** e **CF=1**.

19.8.1 Significato matematico dell'istruzione SHR

Come abbiamo visto nel capitolo dedicato alla matematica del computer, dividere un numero intero senza segno **m**, espresso in base **b**, per un divisore $\mathbf{b^n}$ (**n** intero positivo), equivale a troncare **n** cifre dalla destra dello stesso **m**; in questo modo, tutte le cifre di **m** vengono fatte scorrere di **n** posti verso destra. Risulta, inoltre, che le **n** cifre troncate dalla destra di **m**, rappresentano il resto della divisione intera.
Possiamo dire quindi che una istruzione del tipo:

```
shr ax, n
```

equivale a dividere il contenuto binario di **AX** per $\mathbf{2^n}$.

Come verifica poniamo, **AX=0110101011110010b=27378**, ed eseguiamo l'istruzione:

```
shr ax, 4
```

In presenza di questa istruzione, la **CPU** sottopone il contenuto di **AX** ai seguenti **4** singoli scorrimenti:

```
AX = 0011010101111001b = 13689 = 27378 / 2 (R = 0b), CF = 0

AX = 0001101010111100b = 6844 = 27378 / 4 (R = 10b), CF = 1

AX = 0000110101011110b = 3422 = 27378 / 8 (R = 010b), CF = 0

AX = 0000011010101111b = 1711 = 27378 / 16 (R = 0010b), CF = 0
```

Come si puo' notare, dopo **4** scorrimenti verso destra, il contenuto iniziale **27378** di **AX** risulta diviso per $\mathbf{2^4=16}$; inoltre, i **4** bit traboccati da destra, e cioe', **0010b=2**, rappresentano il resto della divisione intera!

Una **CPU 80486 DX** esegue la precedente istruzione in appena **2** cicli di clock; per ottenere lo stesso risultato, possiamo porre **DX=0**, **AX=0110101011110010b=27378**, **BX=16**, ed eseguire l'istruzione:

```
div bx
```

La stessa **CPU 80486 DX** esegue questa istruzione in **24** cicli di clock, cioe', **12** volte piu' lentamente rispetto all'uso di **SHR**!
Proprio per questo motivo, ogni volta che si deve eseguire una divisione tra numeri interi senza segno, con il divisore esprimibile nella forma $\mathbf{2^n}$ (**n** intero positivo), e' vivamente consigliabile l'utilizzo di **SHR** con contatore **n**.
A differenza di quanto accade con **SHL**, l'istruzione **SHR** fornisce sempre un quoziente e un resto esatti, indipendentemente dal numero di scorrimenti verso destra; ovviamente, ad un certo punto il dividendo diventa **0**, per cui ogni ulteriore scorrimento di **1** bit verso destra, ci fornira' un quoziente **0** e un resto **0**.

In definitiva, una (pseudo) istruzione del tipo:

```
div DEST, 2^n
```

e' sempre sostituibile con una (pseudo) istruzione del tipo:

```
shr DEST, n
```

19.8.2 Shift logical right su operandi di ampiezza arbitraria

Supponiamo di voler far scorrere verso destra, i bit di un numero intero senza segno di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento gia' illustrato per l'istruzione **SHL**.

Vogliamo far scorrere di un posto verso destra, tutti i bit del numero:
```
Num1 = 1011110111110011b
```
Poniamo **AL=0**, e carichiamo in **AH** il byte piu' significativo di **Num1** (**AH=10111101b**); in seguito ad una istruzione:
```
shr ax, 1
```
otteniamo:
```
AX = 0101111010000000b
```
In sostanza, il bit meno significativo di **AH** trabocca da destra, e finisce nel bit piu' significativo di **AL**; si ha quindi **AH=01011110b** e **AL=10000000b**.
Salviamo il contenuto di **AH** in **Num1[1]**, e il contenuto di **AL**, ad esempio, in **BL**; ripetiamo ora il procedimento sul secondo byte (da sinistra) di **Num1**. Poniamo quindi **AL=0** e **AH=11110011b**; in seguito ad una istruzione:
```
shr ax, 1
```
otteniamo:
```
AX = 0111100110000000b
```
In sostanza, il bit meno significativo di **AH** trabocca da destra, e finisce nel bit piu' significativo di **AL**; si ha quindi **AH=01111001b** e **AL=10000000b**.
Il bit che avevamo salvato in **BL** deve essere ora posizionato nel bit piu' significativo di **AH**; a tale proposito, ci basta eseguire l'istruzione:
```
AH = AH OR BL = 01111001b OR 10000000b = 11111001b
```
Salviamo il contenuto di **AH** in **Num1[0]**; a questo punto possiamo notare che:
```
Num1 = 0101111011111001b
```
Il registro **AL** contiene il bit meno significativo di **Num1**, traboccato da destra (nel nostro caso, **AL=10000000b**); appare anche evidente il fatto che, il procedimento appena descritto, puo' essere applicato a numeri interi senza segno di qualunque ampiezza.

Come esempio pratico, consideriamo la seguente definizione:
```
bigNum db 10010010b, 00111100b, 00011111b ; 000111110011110010010010b
```

Il codice che esegue lo scorrimento unitario verso destra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 2 di bigNum

xor      al, al              ; al = 0
mov      ah, BigNum[2]       ; ah = 00011111b
shr      ax, 1               ; ah = 00001111b, al = 10000000b
mov      BigNum[2], ah       ; BigNum[2] = 00001111b
mov      bl, al              ; bl = al = 10000000b

; shifting unitario del BYTE n. 1 di bigNum

xor      al, al              ; al = 0
mov      ah, BigNum[1]       ; ah = 00111100b
shr      ax, 1               ; ah = 00011110b, al = 00000000b
or       ah, bl              ; ah = 10011110b
mov      BigNum[1], ah       ; BigNum[1] = 10011110b
mov      bl, al              ; bl = al = 00000000b

; shifting unitario del BYTE n. 0 di bigNum

xor      al, al              ; al = 0
mov      ah, BigNum[0]       ; ah = 10010010b
shr      ax, 1               ; ah = 01001001b, al = 00000000b
or       ah, bl              ; ah = 01001001b
mov      BigNum[0], ah       ; BigNum[0] = 01001001b
```

Unendo i **3** risultati precedenti, otteniamo:
`BigNum = 000011111001111001001001b`
Il bit meno significativo di **BigNum**, traboccato da destra, si trova memorizzato nel bit piu' significativo di **AL**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.8.3 Effetti provocati da SHR sugli operandi e sui flags

L'esecuzione dell'istruzione **SHR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione al caso che gia' conosciamo, rappresentato da istruzioni del tipo:
`shr cx, cl`

L'esecuzione dell'istruzione **SHR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **SHR** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **SHR** provoca la modifica del bit piu' significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.

I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SHR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.9 L'istruzione SAR

Con il mnemonico **SAR** si indica l'istruzione **Shift Arithmetic Right** (scorrimento aritmetico verso destra); lo scopo di questa istruzione e' quello di effettuare uno scorrimento verso destra, dei bit dell'operando **DEST**, il cui contenuto viene trattato come numero intero con segno in complemento a **2**. Il numero di scorrimenti da effettuare, viene indicato dall'operando **SRC**; i posti rimasti liberi a sinistra nell'operando **DEST**, vengono riempiti con il bit di segno dello stesso **DEST**.
Per ogni singolo scorrimento, il bit meno significativo dell'operando **DEST** trabocca da destra; il valore di tale bit, viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SAR**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SAR** sono le seguenti:

```
SAR   Reg, 1
SAR   Reg, CL
SAR   Reg, Imm8
SAR   Mem, 1
SAR   Mem, CL
SAR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **SAR**, e' formato dal campo **Opcode 110100vw** e dal campo **mod_111_r/m**; se **v=0**, allora l'operando **SRC** e' rappresentato dal valore immediato **1** (un solo scorrimento), mentre se **v=1**, allora l'operando **SRC** e' rappresentato dall'half register **CL** (che contiene il numero di scorrimenti da effettuare). Con le **CPU 80286** e superiori, il numero di scorrimenti puo' essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina e' formato dal campo **Opcode 1100000w** e dal campo **mod_111_r/m**.

Come esempio pratico poniamo **AX=1100011100101011b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
sar ax, cl
```

In presenza di questa istruzione, la **CPU** fa' scorrere di **5** posti verso destra, tutti i bit di **AX**; i **5** posti che si liberano a sinistra, vengono riempiti con il bit di segno (**1**) dello stesso **AX**. La successione dei singoli scorrimenti produce quindi i seguenti risultati:

```
AX = 1110001110010101b, CF = 1

AX = 1111000111001010b, CF = 1

AX = 1111100011100101b, CF = 0

AX = 1111110001110010b, CF = 1

AX = 1111111000111001b, CF = 0
```

I singoli bit che traboccano da destra, vengono salvati nel flag **CF**; nel nostro caso, dopo **5** scorrimenti, l'ultimo bit traboccato da destra e' uno **0**, per cui alla fine otteniamo **AX=1111111000111001b** e **CF=0**.

19.9.1 Significato matematico dell'istruzione SAR

Come e' stato dettagliatamente spiegato nel Capitolo 6, **solamente in determinati casi**, dividere un numero intero con segno **m**, espresso in base **b**, per un divisore $\mathbf{b^n}$ (**n** intero positivo), equivale a troncare **n** cifre dalla destra dello stesso **m**; tale equivalenza sussiste solo se **m** e' un numero intero positivo qualsiasi, oppure se **m** e' un numero intero negativo, multiplo intero del divisore!
Per verificare in pratica questo aspetto, supponiamo di voler lavorare con i numeri interi con segno a **16** bit in complemento a **2**; tali numeri sono quindi compresi tra **-32768** e **+32767**.

Poniamo **AX=0110011001110011b=+26227** (numero intero positivo), ed eseguiamo l'istruzione:
```
sar ax, 4
```
Il risultato prodotto da questa istruzione e':
```
AX = 0000011001100111b = +1639
```
Inoltre, i **4** bit traboccati da destra, rappresentano il valore **0011b=+3**.
Come si puo' notare, i due valori **+1639** e **+3** rappresentano, rispettivamente, il quoziente e il resto della divisione intera (**IDIV**) tra **+26227** e $\mathbf{2^4}$; come sappiamo, in questo caso i risultati coincidono in quanto, **IDIV** produce un quoziente arrotondato verso lo zero, e anche **SAR** produce un valore arrotondato verso lo zero (in sostanza, il valore **+1639.1875** diventa **+1639**, sia per **IDIV** sia per **SAR**).

Poniamo **AX=1110011001110000b=-6544** (numero intero negativo, multiplo intero di **16**), ed eseguiamo l'istruzione:
```
sar ax, 4
```
Il risultato prodotto da questa istruzione e':
```
AX = 1111111001100111b = -409
```
Inoltre, i **4** bit traboccati da destra, rappresentano il valore **0000b=0**.
Come si puo' notare, i due valori **-409** e **0** rappresentano, rispettivamente, il quoziente e il resto della divisione intera (**IDIV**) tra **-6544** e $\mathbf{2^4}$; come sappiamo, in questo caso i risultati coincidono in quanto il resto della divisione e' zero (quoziente esatto).

Poniamo **AX=1000011001110001b=-31119** (numero intero negativo, non divisibile per **16**), ed eseguiamo l'istruzione:
```
sar ax, 4
```
Il risultato prodotto da questa istruzione e':
```
AX = 1111100001100111b = -1945
```
Inoltre, i **4** bit traboccati da destra, rappresentano il valore **0001b=+1**.
Come si puo' notare, il valore **-1945** non coincide con il quoziente **-1944** della divisione intera (**IDIV**) tra **-31119** e $\mathbf{2^4}$; inoltre, il valore **+1** non coincide con il resto **-15** della divisione stessa.
Come sappiamo, in questo caso i risultati non coincidono in quanto, **IDIV** produce un quoziente arrotondato verso lo zero, mentre **SAR** produce un valore arrotondato verso l'infinito negativo; in sostanza, il valore **-1944.9375** diventa **-1944** per **IDIV**, e **-1945** per **SAR**!

In base alle considerazioni appena esposte, si sconsiglia vivamente l'utilizzo di **SAR** per eseguire divisioni rapide tra un numero intero con segno e un divisore del tipo $\mathbf{2^n}$; infatti, il rischio di ottenere risultati sbagliati e' troppo elevato.
In questi casi, conviene utilizzare **IDIV** che fornisce sempre il risultato corretto; il prezzo da pagare consiste, naturalmente, in una minore velocita' di esecuzione.

19.9.2 Shift arithmetic right su operandi di ampiezza arbitraria

Supponiamo di voler far scorrere verso destra, i bit di un numero intero con segno di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento gia' illustrato per l'istruzione **SHR**.
Rispetto all'esempio illustrato per **SHR**, l'unica novita' e' data dal fatto che nello shift aritmetico verso destra e' necessario preservare il bit di segno dell'operando; a tale proposito, non dobbiamo fare altro che utilizzare **SAR** per lo shifting del byte piu' significativo dell'operando, e **SHR** per lo shifting degli altri byte!

Come esempio pratico, consideriamo la seguente definizione:
`bigNum db 10010010b, 00111100b, 11011111b ; 110111110011110010010010b`

Il codice che esegue lo scorrimento unitario verso destra, assume il seguente aspetto:

```
; shifting unitario aritmetico del BYTE n. 2 di bigNum

xor      al, al              ; al = 0
mov      ah, BigNum[2]       ; ah = 11011111b
sar      ax, 1               ; ah = 11101111b, al = 10000000b
mov      BigNum[2], ah       ; BigNum[2] = 11101111b
mov      bl, al              ; bl = al = 10000000b

; shifting unitario logico del BYTE n. 1 di bigNum

xor      al, al              ; al = 0
mov      ah, BigNum[1]       ; ah = 00111100b
shr      ax, 1               ; ah = 00011110b, al = 00000000b
or       ah, bl              ; ah = 10011110b
mov      BigNum[1], ah       ; BigNum[1] = 10011110b
mov      bl, al              ; bl = al = 00000000b

; shifting unitario logico del BYTE n. 0 di bigNum

xor      al, al              ; al = 0
mov      ah, BigNum[0]       ; ah = 10010010b
shr      ax, 1               ; ah = 01001001b, al = 00000000b
or       ah, bl              ; ah = 01001001b
mov      BigNum[0], ah       ; BigNum[0] = 01001001b
```

Unendo i **3** risultati precedenti, otteniamo:
`BigNum = 111011111001111001001001b`
Il bit meno significativo di **BigNum**, traboccato da destra, si trova memorizzato nel bit piu' significativo di **AL**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.9.3 Effetti provocati da SAR sugli operandi e sui flags

L'esecuzione dell'istruzione **SAR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione al caso che gia' conosciamo, rappresentato da istruzioni del tipo:

```
sar cx, cl
```

L'esecuzione dell'istruzione **SAR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **SAR** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, nel caso di uno scorrimento unitario aritmetico verso destra, la **CPU** pone sempre **OF=0**; infatti, l'istruzione **SAR** preserva sempre il bit di segno dell'operando **DEST**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SAR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.10 Le istruzioni SHLD e SHRD

Con le **CPU 80386** e superiori, vengono rese disponibili due ulteriori istruzioni di shifting, rappresentate dai mnemonici **SHLD** e **SHRD**; tali istruzioni sono molto simili a **SHL** e **SHR**, ma rispetto a queste ultime, svolgono un lavoro molto piu' sofisticato.
Entrambe le istruzioni richiedono tre operandi; le cifre del primo operando (**DEST**) vengono sottoposte ad un numero di scorrimenti indicato dal terzo operando (contatore). I posti che si liberano nell'operando **DEST**, vengono riempiti con altrettanti bit prelevati dal secondo operando (**SRC**); il contenuto del secondo operando rimane inalterato.
Il termine "doppia precisione" si riferisce al fatto che con queste istruzioni, possiamo effettuare operazioni di shifting su un operando **SRC** a **64** bit (come, ad esempio, una coppia di registri a **32** bit del tipo **EDX:EAX**).

Come al solito, vengono presi in considerazione solamente i **5** bit meno significativi del contatore; questa volta, pero', il contenuto del contatore non viene modificato (in sostanza, la **CPU** si serve di un registro temporaneo, nel quale viene memorizzato il contenuto dei primi **5** bit del contatore).

19.10.1 L'istruzione SHLD

Con il mnemonico **SHLD** si indica l'istruzione **SHift logical Left, Double precision** (scorrimento logico verso sinistra, in doppia precisione); lo scopo di questa istruzione e' quello di effettuare uno scorrimento logico verso sinistra, dei bit dell'operando **DEST** (primo operando). Il numero di scorrimenti da effettuare, viene indicato dal contenuto del terzo operando (contatore); i posti rimasti liberi a destra nell'operando **DEST**, vengono riempiti con altrettanti bit fatti traboccare dalla sinistra dell'operando **SRC** (secondo operando). Il contenuto dell'operando **SRC** non subisce nessuna

modifica.
Per ogni singolo scorrimento, il bit piu' significativo dell'operando **DEST** trabocca da sinistra; il valore di tale bit, viene salvato nel flag **CF**.
I tre operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **SHLD**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHLD** sono le seguenti:

```
SHLD  Reg16, Reg16, Imm8
SHLD  Reg16, Reg16, CL
SHLD  Mem16, Reg16, Imm8
SHLD  Mem16, Reg16, CL
SHLD  Reg32, Reg32, Imm8
SHLD  Reg32, Reg32, CL
SHLD  Mem32, Reg32, Imm8
SHLD  Mem32, Reg32, CL
```

Come esempio pratico poniamo **EAX=11010111001010110111000111001010b**, **EBX=00111011000111111001001111000010b**, **CL=4**, ed eseguiamo la seguente istruzione:
```
shld eax, ebx, cl
```
In presenza di questa istruzione, la **CPU** fa' scorrere di **4** posti verso sinistra, tutti i bit di **EAX**; i **4** posti che si liberano a destra, vengono riempiti con altrettanti bit fatti traboccare dalla sinistra del registro **EBX**. Alla fine otteniamo il seguente risultato:
```
EAX = 01110010101101110001110010100011b, EBX=00111011000111111001001111000010b,
CF = 1
```
Come si puo' notare, il contenuto di **EBX** rimane inalterato; inoltre, l'ultimo bit traboccato dalla sinistra di **EAX** e' un **1**, per cui alla fine otteniamo **CF=1**.

Nei precedenti capitoli abbiamo visto che determinate istruzioni della **CPU**, richiedono che uno dei loro operandi sia rappresentato, obbligatoriamente, dalla coppia **DX:AX**; si presenta quindi spesso la necessita' di salvare in **DX:AX**, un valore a **32** bit (con **DX** che deve contenere la **WORD** piu' significativa).
Supponendo che il valore a **32** bit sia memorizzato in **EAX**, possiamo allora procedere in questo modo:

```
push     eax                  ; salva eax nello stack
pop      ax                   ; ax = word meno significativa
pop      dx                   ; dx = word piu' significativa
```

Una soluzione molto piu' semplice consiste, invece, nello scrivere l'istruzione:
```
shld edx, eax, 16
```

19.10.2 L'istruzione SHRD

Con il mnemonico **SHRD** si indica l'istruzione **SHift logical Right, Double precision** (scorrimento logico verso destra, in doppia precisione); lo scopo di questa istruzione e' quello di effettuare uno scorrimento logico verso destra, dei bit dell'operando **DEST** (primo operando). Il numero di scorrimenti da effettuare, viene indicato dal contenuto del terzo operando (contatore); i posti rimasti liberi a sinistra nell'operando **DEST**, vengono riempiti con altrettanti bit fatti traboccare dalla destra dell'operando **SRC** (secondo operando). Il contenuto dell'operando **SRC** non subisce nessuna modifica.
Per ogni singolo scorrimento, il bit meno significativo dell'operando **DEST** trabocca da destra; il valore di tale bit, viene salvato nel flag **CF**.
I tre operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da

SHRD, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **SHRD** sono le seguenti:

```
SHRD  Reg16, Reg16, Imm8
SHRD  Reg16, Reg16, CL
SHRD  Mem16, Reg16, Imm8
SHRD  Mem16, Reg16, CL
SHRD  Reg32, Reg32, Imm8
SHRD  Reg32, Reg32, CL
SHRD  Mem32, Reg32, Imm8
SHRD  Mem32, Reg32, CL
```

Come esempio pratico poniamo **EAX=00000111001010110111000111001010b**, **EBX=00111011000111111001001111011111b**, **CL=4**, ed eseguiamo la seguente istruzione:

```
shrd eax, ebx, cl
```

In presenza di questa istruzione, la **CPU** fa' scorrere di **4** posti verso destra, tutti i bit di **EAX**; i **4** posti che si liberano a sinistra, vengono riempiti con altrettanti bit fatti traboccare dalla destra del registro **EBX**. Alla fine otteniamo il seguente risultato:

```
EAX=11110000011100101011011100011100b, EBX=00111011000111111001001111011111b, CF
= 1
```

Come si puo' notare, il contenuto di **EBX** rimane inalterato; inoltre, l'ultimo bit traboccato dalla destra di **EAX** e' un **1**, per cui alla fine otteniamo **CF=1**.

19.10.3 Effetti provocati da SHLD e SHRD, sugli operandi e sui flags

L'esecuzione delle istruzioni **SHLD** e **SHRD**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dalle istruzioni stesse; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione ai casi rappresentati da istruzioni del tipo:

```
shld ecx, ecx, cl
```

Si ricordi che in presenza delle istruzioni **SHLD** e **SHRD**, la **CPU** utilizza come contatore il contenuto dei primi **5** bit di **CL**, senza apportare nessuna modifica allo stesso **CL**; ne consegue che la precedente istruzione, non provoca nessuna modifica sul contenuto del registro destinazione **ECX**!

L'esecuzione delle istruzioni **SHLD** e **SHRD** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione delle istruzioni **SHLD** e **SHRD** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1**; se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, la **CPU** pone **OF=1** quando le istruzioni **SHLD** e **SHRD** provocano la modifica del bit piu' significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato da **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **SHLD** e **SHRD**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

A differenza di quanto accade con **SHL**, **SAL**, **SHR** e **SAR**, se il contatore specifica un numero eccessivo di scorrimenti, le istruzioni **SHLD** e **SHRD** forniscono risultati privi di senso; cio'

accade, ad esempio, con una istruzione del tipo:
```
shrd ax, bx, 30
```
In un caso del genere, anche i flags assumono valori privi di significato!

19.11 L'istruzione ROL

Con il mnemonico **ROL** si indica l'istruzione **ROtate Left** (rotazione verso sinistra); lo scopo di questa istruzione e' quello di effettuare una rotazione verso sinistra, dei bit dell'operando **DEST**. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.
Per ogni singola rotazione, il bit piu' significativo dell'operando **DEST** trabocca da sinistra; tale bit viene salvato nel flag **CF**, e contemporaneamente, viene fatto rientrare dalla destra dello stesso operando **DEST**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **ROL**, viene memorizzato nell'operando **DEST**.

Le uniche forme lecite per l'istruzione **ROL** sono le seguenti:

```
ROL   Reg, 1
ROL   Reg, CL
ROL   Reg, Imm8
ROL   Mem, 1
ROL   Mem, CL
ROL   Mem, Imm8
```

Il codice macchina generale per l'istruzione **ROL**, e' formato dal campo **Opcode 110100vw** e dal campo **mod_000_r/m**; se **v=0**, allora l'operando **SRC** e' rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** e' rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni puo' essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina e' formato dal campo **Opcode 1100000w** e dal campo **mod_000_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:
```
rol ax, cl
```
In presenza di questa istruzione, la **CPU** fa' ruotare di **5** posti verso sinistra, tutti i bit di **AX**; ogni singolo bit che trabocca dalla sinistra di **AX**, viene memorizzato in **CF**, e contemporaneamente, viene fatto rientrare dalla destra dello stesso **AX**. La successione delle singole rotazioni produce quindi i seguenti risultati:

```
AX = 1110010110111000b, CF = 0

AX = 1100101101110001b, CF = 1

AX = 1001011011100011b, CF = 1

AX = 0010110111000111b, CF = 1

AX = 0101101110001110b, CF = 0
```

L'ultimo bit sottoposto a rotazione e' uno **0**; di conseguenza, alla fine otteniamo **AX=0101101110001110b** e **CF=0**.

19.11.1 ROL su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso sinistra, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento gia' illustrato per l'istruzione **SHL**.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 10011111b ; 100111110011110010010010b
```

Il codice che esegue la rotazione unitaria verso sinistra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 0 di bigNum

xor      ah, ah               ; ah = 0
mov      al, BigNum[0]        ; al = 10010010b
shl      ax, 1                ; ah = 00000001b, al = 00100100b
mov      BigNum[0], al        ; BigNum[0] = 00100100b
mov      bl, ah               ; bl = ah = 00000001b

; shifting unitario del BYTE n. 1 di bigNum

xor      ah, ah               ; ah = 0
mov      al, BigNum[1]        ; al = 00111100b
shl      ax, 1                ; ah = 00000000b, al = 01111000b
or       al, bl               ; al = 01111001b
mov      BigNum[1], al        ; BigNum[1] = 01111001b
mov      bl, ah               ; bl = ah = 00000000b

; shifting unitario del BYTE n. 2 di bigNum

xor      ah, ah               ; ah = 0
mov      al, BigNum[2]        ; al = 10011111b
shl      ax, 1                ; ah = 00000001b, al = 00111110b
or       al, bl               ; al = 00111110b
mov      BigNum[2], al        ; BigNum[2] = 00111110b

; rotazione del bit piu' significativo

or       BigNum[0], ah        ; BigNum[0] = 00100101b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 001111100111100100100101b
```

Il bit piu' significativo di **BigNum**, traboccato da sinistra, oltre ad essere rientrato da destra, si trova anche memorizzato nel bit meno significativo di **AH**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.11.2 Effetti provocati da ROL sugli operandi e sui flags

L'esecuzione dell'istruzione **ROL**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione al caso che gia' conosciamo, rappresentato da istruzioni del tipo:

```
rol cx, cl
```

L'esecuzione dell'istruzione **ROL** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **ROL** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **ROL** provoca la modifica del bit piu' significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla sinistra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **ROL**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.12 L'istruzione RCL

Con il mnemonico **RCL** si indica l'istruzione **Rotate through Carry Left** (rotazione verso sinistra attraverso **CF**); lo scopo di questa istruzione e' quello di effettuare una rotazione verso sinistra, dei bit dell'operando **DEST+CF**, con **CF** che ricopre il ruolo di bit meno significativo. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.
Per ogni singola rotazione, il contenuto corrente di **CF** viene fatto entrare dalla destra di **DEST**, provocando lo scorrimento di una posizione verso sinistra, di tutti i bit dello stesso **DEST**; il bit piu' significativo di **DEST** trabocca da sinistra, e viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **RCL**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **RCL** sono le seguenti:

```
RCL   Reg, 1
RCL   Reg, CL
RCL   Reg, Imm8
RCL   Mem, 1
RCL   Mem, CL
RCL   Mem, Imm8
```

Il codice macchina generale per l'istruzione **RCL**, e' formato dal campo **Opcode 110100vw** e dal campo **mod_010_r/m**; se **v=0**, allora l'operando **SRC** e' rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** e' rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni puo' essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina e' formato dal campo

Opcode 1100000w e dal campo **mod_010_r/m**.
Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, **CF=1**, ed eseguiamo la seguente istruzione:

```
rcl ax, cl
```

In presenza di questa istruzione, la **CPU** fa' ruotare di **5** posti verso sinistra, tutti i bit di **AX+CF**; in sostanza, stiamo sottoponendo a rotazione verso sinistra, un operando a **17** bit, con **CF** che ricopre il ruolo di bit meno significativo. La successione delle singole rotazioni produce quindi i seguenti risultati:

```
AX = 1110010110111001b, CF = 0

AX = 1100101101110010b, CF = 1

AX = 1001011011100101b, CF = 1

AX = 0010110111001011b, CF = 1

AX = 0101101110010111b, CF = 0
```

L'ultimo bit traboccato da sinistra e' uno **0**; di conseguenza, alla fine otteniamo **AX=0101101110010111b** e **CF=0**.

19.12.1 RCL su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso sinistra, con l'ausilio di **CF**, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento gia' illustrato per l'istruzione **SHL**.
Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010010b, 00111100b, 10011111b ; 100111110011110010010010b
```

Assumendo di avere, inizialmente, **CF=0**, possiamo scrivere il seguente codice:

```
; shifting unitario del BYTE n. 0 di bigNum

pushf                       ; preserva i flags (CF = 0)
xor     ah, ah              ; ah = 0
mov     al, BigNum[0]       ; al = 10010010b
popf                        ; ripristina i flags (CF = 0)
rcl     ax, 1               ; ah = 00000001b, al = 00100100b
mov     BigNum[0], al       ; BigNum[0] = 00100100b
mov     bl, ah              ; bl = ah = 00000001b

; shifting unitario del BYTE n. 1 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[1]       ; al = 00111100b
shl     ax, 1               ; ah = 00000000b, al = 01111000b
or      al, bl              ; al = 01111001b
mov     BigNum[1], al       ; BigNum[1] = 01111001b
mov     bl, ah              ; bl = ah = 00000000b

; shifting unitario del BYTE n. 2 di bigNum

xor     ah, ah              ; ah = 0
mov     al, BigNum[2]       ; al = 10011111b
shl     ax, 1               ; ah = 00000001b, al = 00111110b
or      al, bl              ; al = 00111110b
mov     BigNum[2], al       ; BigNum[2] = 00111110b
sahf                        ; SF,ZF,-,AF,-,PF,-,CF = 00000001b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 001111100111100100100100b, CF = 1
```

Il vecchio contenuto di **CF**, e cioe' **0**, viene fatto entrare dalla destra di **BigNum**; tutti i bit di **BigNum** scorrono di una posizione verso sinistra. Il bit piu' significativo di **BigNum**, trabocca da sinistra e finisce nel bit meno significativo di **AH**; a questo punto, l'istruzione **SAHF** pone **CF=1** (infatti, **CF** occupa la posizione n. **0** nel registro **FLAGS**).

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.12.2 Effetti provocati da RCL sugli operandi e sui flags

L'esecuzione dell'istruzione **RCL**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione al caso che gia' conosciamo, rappresentato da istruzioni del tipo:

```
rcl cx, cl
```

L'esecuzione dell'istruzione **RCL** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **RCL** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **RCL** provoca la modifica del bit piu' significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla sinistra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **RCL**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.13 L'istruzione ROR

Con il mnemonico **ROR** si indica l'istruzione **ROtate Right** (rotazione verso destra); lo scopo di questa istruzione e' quello di effettuare una rotazione verso destra, dei bit dell'operando **DEST**. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.
Per ogni singola rotazione, il bit meno significativo dell'operando **DEST** trabocca da destra; tale bit viene salvato nel flag **CF**, e contemporaneamente, viene fatto rientrare dalla sinistra dello stesso operando **DEST**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **ROR**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **ROR** sono le seguenti:

```
ROR   Reg, 1
ROR   Reg, CL
ROR   Reg, Imm8
ROR   Mem, 1
ROR   Mem, CL
ROR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **ROR**, e' formato dal campo **Opcode 110100vw** e dal campo **mod_001_r/m**; se **v=0**, allora l'operando **SRC** e' rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** e' rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni puo' essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina e' formato dal campo **Opcode 1100000w** e dal campo **mod_001_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, ed eseguiamo la seguente istruzione:

```
ror ax, cl
```

In presenza di questa istruzione, la **CPU** fa' ruotare di **5** posti verso destra, tutti i bit di **AX**; ogni singolo bit che trabocca dalla destra di **AX**, viene memorizzato in **CF**, e contemporaneamente, viene fatto rientrare dalla sinistra dello stesso **AX**. La successione delle singole rotazioni produce quindi i seguenti risultati:

```
AX = 0011100101101110b, CF = 0

AX = 0001110010110111b, CF = 0

AX = 1000111001011011b, CF = 1

AX = 1100011100101101b, CF = 1

AX = 1110001110010110b, CF = 1
```

L'ultimo bit sottoposto a rotazione e' un **1**; di conseguenza, alla fine otteniamo **AX=1110001110010110b** e **CF=1**.

19.13.1 ROR su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso destra, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento gia' illustrato per l'istruzione **SHR**.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010011b, 00111100b, 00011111b ; 000111110011110010010011b
```

Il codice che esegue la rotazione unitaria verso destra, assume il seguente aspetto:

```
; shifting unitario del BYTE n. 2 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[2]       ; ah = 00011111b
shr     ax, 1               ; ah = 00001111b, al = 10000000b
mov     BigNum[2], ah       ; BigNum[2] = 00001111b
mov     bl, al              ; bl = al = 10000000b

; shifting unitario del BYTE n. 1 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[1]       ; ah = 00111100b
shr     ax, 1               ; ah = 00011110b, al = 00000000b
or      ah, bl              ; ah = 10011110b
mov     BigNum[1], ah       ; BigNum[1] = 10011110b
mov     bl, al              ; bl = al = 00000000b

; shifting unitario del BYTE n. 0 di bigNum

xor     al, al              ; al = 0
mov     ah, BigNum[0]       ; ah = 10010011b
shr     ax, 1               ; ah = 01001001b, al = 10000000b
or      ah, bl              ; ah = 01001001b
mov     BigNum[0], ah       ; BigNum[0] = 01001001b

; rotazione del bit meno significativo

or      BigNum[2], al       ; BigNum[2] = 10001111b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 100011111001111001001001b
```

Il bit meno significativo di **BigNum**, traboccato da destra, oltre ad essere rientrato da sinistra, si trova anche memorizzato nel bit piu' significativo di **AL**.

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.13.2 Effetti provocati da ROR sugli operandi e sui flags

L'esecuzione dell'istruzione **ROR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione al caso che gia' conosciamo, rappresentato da istruzioni del tipo:

```
ror cx, cl
```

L'esecuzione dell'istruzione **ROR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **ROR** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.

Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **ROR** provoca la modifica del bit piu' significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **ROR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.14 L'istruzione RCR

Con il mnemonico **RCR** si indica l'istruzione **Rotate through Carry Right** (rotazione verso destra attraverso **CF**); lo scopo di questa istruzione e' quello di effettuare una rotazione verso destra, dei bit dell'operando **CF+DEST**, con **CF** che ricopre il ruolo di bit piu' significativo. Il numero di rotazioni da effettuare, viene indicato dall'operando **SRC**.
Per ogni singola rotazione, il contenuto corrente di **CF** viene fatto entrare dalla sinistra di **DEST**, provocando lo scorrimento di una posizione verso destra, di tutti i bit dello stesso **DEST**; il bit meno significativo di **DEST** trabocca da destra, e viene salvato nel flag **CF**.
Entrambi gli operandi devono essere specificati, esplicitamente, dal programmatore; il risultato prodotto da **RCR**, viene memorizzato nell'operando **DEST**.
Le uniche forme lecite per l'istruzione **RCR** sono le seguenti:

```
RCR   Reg, 1
RCR   Reg, CL
RCR   Reg, Imm8
RCR   Mem, 1
RCR   Mem, CL
RCR   Mem, Imm8
```

Il codice macchina generale per l'istruzione **RCR**, e' formato dal campo **Opcode 110100vw** e dal campo **mod_011_r/m**; se **v=0**, allora l'operando **SRC** e' rappresentato dal valore immediato **1** (una sola rotazione), mentre se **v=1**, allora l'operando **SRC** e' rappresentato dall'half register **CL** (che contiene il numero di rotazioni da effettuare). Con le **CPU 80286** e superiori, il numero di rotazioni puo' essere indicato anche attraverso un **Imm8**; in tal caso, il codice macchina e' formato dal campo **Opcode 1100000w** e dal campo **mod_011_r/m**.

Come esempio pratico poniamo **AX=0111001011011100b**, **CL=5**, **CF=1**, ed eseguiamo la seguente istruzione:

```
rcr ax, cl
```

In presenza di questa istruzione, la **CPU** fa' ruotare di **5** posti verso destra, tutti i bit di **CF+AX**; in sostanza, stiamo sottoponendo a rotazione verso destra, un operando a **17** bit, con **CF** che ricopre il ruolo di bit piu' significativo. La successione delle singole rotazioni produce quindi i seguenti risultati:

```
AX = 1011100101101110b, CF = 0

AX = 0101110010110111b, CF = 0

AX = 0010111001011011b, CF = 1
```

```
AX = 1001011100101101b, CF = 1

AX = 1100101110010110b, CF = 1
```

L'ultimo bit traboccato da destra e' un **1**; di conseguenza, alla fine otteniamo **AX=1100101110010110b** e **CF=1**.

19.14.1 RCR su operandi di ampiezza arbitraria

Supponiamo di voler far ruotare verso destra, con l'ausilio di **CF**, i bit di un numero binario di ampiezza arbitraria; in tal caso, possiamo facilmente adattare il procedimento gia' illustrato per l'istruzione **SHR**.

Come esempio pratico, consideriamo la seguente definizione:

```
bigNum db 10010011b, 00111100b, 10011111b ; 100111110011110010010011b
```

Assumendo di avere, inizialmente, **CF=0**, possiamo scrivere il seguente codice:

```
; shifting unitario del BYTE n. 2 di bigNum

pushf                           ; preserva i flags (CF = 0)
xor      al, al                 ; al = 0
mov      ah, BigNum[2]          ; ah = 10011111b
popf                            ; ripristina i flags (CF = 0)
rcr      ax, 1                  ; ah = 01001111b, al = 10000000b
mov      BigNum[2], ah          ; BigNum[2] = 01001111b
mov      bl, al                 ; bl = al = 10000000b

; shifting unitario del BYTE n. 1 di bigNum

xor      al, al                 ; al = 0
mov      ah, BigNum[1]          ; ah = 00111100b
shr      ax, 1                  ; ah = 00011110b, al = 00000000b
or       ah, bl                 ; ah = 10011110b
mov      BigNum[1], ah          ; BigNum[1] = 10011110b
mov      bl, al                 ; bl = al = 00000000b

; shifting unitario del BYTE n. 0 di bigNum

xor      al, al                 ; al = 0
mov      ah, BigNum[0]          ; ah = 10010011b
shr      ax, 1                  ; ah = 01001001b, al = 10000000b
or       ah, bl                 ; ah = 01001001b
mov      BigNum[0], ah          ; BigNum[0] = 01001001b

mov      ah, al                 ; ah = 10000000b
shr      ah, 7                  ; ah = 00000001b
sahf                            ; SF,ZF,-,AF,-,PF,-,CF = 00000001b
```

Unendo i **3** risultati precedenti, otteniamo:

```
BigNum = 010011111001111001001001b, CF = 1
```

Il vecchio contenuto di **CF**, e cioe' **0**, viene fatto entrare dalla sinistra di **BigNum**; tutti i bit di **BigNum** scorrono di una posizione verso destra. Il bit meno significativo di **BigNum**, trabocca da destra e finisce nel bit piu' significativo di **AL**; a questo punto, spostiamo **AL** in **AH**, facciamo scorrere di **7** posti verso destra il contenuto di **AH**, ed eseguiamo una istruzione **SAHF** che pone **CF=1** (infatti, **CF** occupa la posizione n. **0** nel registro **FLAGS**).

Se vogliamo visualizzare il contenuto binario di **BigNum**, possiamo procedere nel solito modo con l'ausilio di **writeBin8**; in questo caso, l'output deve iniziare da **BigNum[2]**.

19.14.2 Effetti provocati da RCR sugli operandi e sui flags

L'esecuzione dell'istruzione **RCR**, modifica il contenuto del solo operando **DEST** che viene sovrascritto dal risultato prodotto dall'istruzione stessa; il contenuto dell'operando **SRC** rimane inalterato.
Bisogna pero' prestare particolare attenzione al caso che gia' conosciamo, rappresentato da istruzioni del tipo:

```
rcr cx, cl
```

L'esecuzione dell'istruzione **RCR** con **CL=0** o **Imm8=0**, non modifica nessun campo del **Flags Register**; ovviamente, in un caso del genere, il contenuto di **DEST** rimane inalterato.
L'esecuzione dell'istruzione **RCR** con contatore non nullo, modifica i campi **CF**, **PF**, **AF**, **ZF**, **SF** e **OF** del **Flags Register**; il solo campo **AF** assume un valore indeterminato, e non ha quindi nessun significato.
Il campo **OF** assume un valore sensato solo quando il contatore vale **1** (e quindi anche con **CL=1** o **Imm8=1**); se il contatore e' maggiore di **1**, il flag **OF** assume un valore indeterminato, e non ha quindi nessun significato. Come si puo' facilmente immaginare, la **CPU** pone **OF=1** quando l'istruzione **RCR** provoca la modifica del bit piu' significativo (bit di segno) di **DEST**; in caso contrario, la **CPU** pone **OF=0**.
Il campo **CF** (nel caso in cui il contatore sia diverso da zero), contiene l'ultimo bit traboccato dalla destra di **DEST**.
I flags **PF**, **ZF** e **SF** (nel caso in cui il contatore sia diverso da zero) forniscono informazioni ordinarie relative al risultato prodotto da **RCR**; in sostanza, **PF** indica se i primi **8** bit del risultato contengono un numero pari o dispari di bit a livello logico **1**, **ZF** indica se il risultato vale zero, **SF** indica se il bit piu' significativo del risultato vale **1**.

19.15 Istruzioni logiche nei linguaggi di alto livello

Molti linguaggi di alto livello, forniscono due categorie di istruzioni logiche, chiamate **operatori logici** e **operatori booleani**; gli operatori logici agiscono sui singoli bit dei loro operandi (proprio come succede con le istruzioni della **CPU** esaminate in questo capitolo), mentre gli operatori booleani agiscono sul cosiddetto **valore booleano** dei loro operandi.
Il valore booleano indica se il contenuto di un operando e' uguale o diverso da zero; un operando e' **TRUE** (o **1**) se il suo contenuto e' diverso da zero, mentre e' **FALSE** (o **0**) se il suo contenuto e' uguale a zero.

Se **A** e **B** sono due operandi di tipo numerico intero, allora una espressione del tipo:

```
C = A AND B
```

viene definita **espressione logica**; il risultato che viene memorizzato in **C** e' un valore numerico intero, conseguenza di un **AND logico** (bit per bit) tra i due operandi **A** e **B**.

Se **A** e **B** sono due operandi di tipo booleano (**TRUE** o **FALSE**), allora una espressione del tipo:

```
C = A AND B
```

viene definita **espressione booleana**; il risultato che viene memorizzato in **C** e' un valore booleano, conseguenza di un **AND booleano** tra i due operandi **A** e **B**. Il risultato e' **TRUE** se, e solo se, entrambi gli operandi sono **TRUE**.

Osserviamo allora che se **A=10101010b** e **B=01010101b**, un **AND** logico produce:

```
A AND B = 10101010b AND 01010101b = 00000000b = 0
```

Un **AND** booleano produce, invece:
`A AND B = TRUE AND TRUE = TRUE = 1`
(sia **A** che **B** hanno, infatti, un valore non nullo).

Nel linguaggio **C/C++**, le due categorie di istruzioni logiche utilizzano nomi differenti; tali nomi vengono illustrati in Figura 5.

Figura 5 - Operatori logici e booleani del C/C++

Operatori logici		**Operatori booleani**	
Nome	**Significato**	**Nome**	**Significato**
&	AND logico	&&	AND booleano
~	NOT logico		
|	OR logico inclusivo	||	OR booleano inclusivo
^	OR logico esclusivo		
<<	SHL logico		
>>	SHR logico		

In **C/C++** la condizione **TRUE** viene indicata dal valore numerico **1** (o da un qualsiasi valore diverso da zero); invece, la condizione **FALSE** viene indicata dal valore numerico **0**.
Gli operatori logici possono essere applicati solamente a variabili di tipo intero; gli operatori booleani possono essere applicati a qualsiasi variabile numerica.
Per capire la differenza di comportamento tra gli operatori logici e booleani del **C/C++**, possiamo eseguire il seguente programma **C**:

```
/* file bool.c */

#include < stdio.h >

unsigned int   v1 = 0x0F0F;   /* 0000111100001111b */
unsigned int   v2 = 0xF0F0;   /* 1111000011110000b */

int main(void)
{
   printf("Operatori logici:\n");
   printf("v1 or v2  = %X\n", v1 | v2);
   printf("v1 and v2 = %X\n", v1 & v2);
   printf("Operatori booleani:\n");
   printf("v1 or v2  = %u\n", v1 || v2);
   printf("v1 and v2 = %u\n", v1 && v2);

   return 0;
}
```

Nel linguaggio **Pascal/Delphi**, le due categorie di istruzioni logiche utilizzano gli stessi nomi, e vengono distinte dal compilatore in base al contesto in cui compaiono; tali nomi vengono illustrati in Figura 6.

Figura 6 - Operatori logici e booleani del Pascal/Delphi

Operatori logici		Operatori booleani	
Nome	**Significato**	**Nome**	**Significato**
and	AND logico	and	AND booleano
not	NOT logico	not	NOT booleano
or	OR logico inclusivo	or	OR booleano inclusivo
xor	OR logico esclusivo	xor	OR booleano esclusivo
shl	SHL logico		
shr	SHR logico		

In **Pascal/Delphi** la condizione **TRUE** viene indicata dal nome riservato **true**; invece, la condizione **FALSE** viene indicata dal nome riservato **false**.
Gli operatori logici possono essere applicati solamente a variabili di tipo intero; gli operatori booleani possono essere applicati solamente a variabili di tipo booleano.
Per capire la differenza di comportamento tra gli operatori logici e booleani del **Pascal/Delphi**, possiamo eseguire il seguente programma **Pascal**:

```
{ file bool.pas }

program Bool;

var
   v1, v2:     Word;
   b1, b2:     Boolean;

begin
   WriteLn('Operatori logici:');
   v1 := $0F0F;                        { 0000111100001111b }
   v2 := $F0F0;                        { 1111000011110000b }
   WriteLn('v1 or v2  = ', v1 or v2);
   WriteLn('v1 and v2 = ', v1 and v2);
   WriteLn('Operatori booleani:');
   b1 := true;
   b2 := false;
   WriteLn('b1 or b2  = ', b1 or b2);
   WriteLn('b1 and b2 = ', b1 and b2);
end.
```

Come vedremo in un apposito capitolo, anche gli assembler come **MASM** e **TASM** forniscono una serie di operatori logici e booleani; tali operatori, non hanno niente a che vedere con le analoghe istruzioni della **CPU**.

19.16 Applicazioni pratiche delle istruzioni logiche

Come applicazione pratica delle istruzioni logiche della **CPU**, possiamo riprendere l'esempio del precedente capitolo, relativo alla conversione in binario di un numero **Unpacked BCD**; abbiamo visto che applicando il metodo delle divisioni successive (con divisore **2**) al numero **07050301h** (in formato **Unpacked BCD**), otteniamo la seguente sequenza di resti:

```
R0=01h, R1=01h, R2=00h, R3=01h, R4=00h, R5=01h, R6=01h
R7=00h, R8=01h, R9=00h, R10=01h, R11=01h, R12=01h
```

I vari resti rappresentano le cifre binarie del numero decimale **7531**; come facciamo a memorizzare tali resti nei singoli bit di un numero binario?

Adesso che abbiamo a disposizione le istruzioni logiche, possiamo rispondere a questa domanda in modo molto semplice; come si puo' facilmente intuire, possiamo risolvere il problema con l'ausilio delle sole istruzioni **OR** e **SHL**!

Prima di tutto, predisponiamo le seguenti definizioni:

```
resto       db    01h, 01h, 00h, 01h, 00h, 01h, 01h
                  00h, 01h, 00h, 01h, 01h, 01h
bcd2binary  dw    0
```

A questo punto, possiamo scrivere il seguente codice:

```
mov      al, resto[0]               ; al = 00000001b
or       byte ptr bcd2binary[0], al ; bcd2binary = 0000000000000001b

mov      al, resto[1]               ; al = 00000001b
shl      al, 1                      ; al = 00000010b
or       byte ptr bcd2binary[0], al ; bcd2binary = 0000000000000011b

mov      al, resto[2]               ; al = 00000000b
shl      al, 2                      ; al = 00000000b
or       byte ptr bcd2binary[0], al ; bcd2binary = 0000000000000011b

mov      al, resto[3]               ; al = 00000001b
shl      al, 3                      ; al = 00001000b
or       byte ptr bcd2binary[0], al ; bcd2binary = 0000000000001011b

; ... e cosi' via
```

Dopo aver riempito gli **8** bit meno significativi di **bcd2binary**, dobbiamo procedere allo stesso modo sulla variabile **bcd2binary[1]**.

Capitolo 20 - Istruzioni per il trasferimento del controllo

I primi calcolatori elettronici realizzati negli anni **50**, erano dotati di CPU molto elementari che potevano eseguire i programmi solo in modo sequenziale; i programmi erano formati da una sequenza di istruzioni numerate **1**, **2**, **3**, etc, che venivano eseguite in successione dalla CPU senza la possibilita' di saltare da una parte all'altra del codice. L'esecuzione partiva dall'istruzione n. **1** e proseguiva ordinatamente con l'istruzione n. **2**, l'istruzione n. **3**, etc; ogni volta che la CPU caricava una nuova istruzione da eseguire, l'**instruction pointer** veniva aggiornato in modo da puntare all'istruzione immediatamente successiva. Come si puo' facilmente intuire, i programmi sequenziali presentano una struttura estremamente semplice che consente alla CPU di eseguire le varie istruzioni in modo relativamente rapido; lo svantaggio evidente dei programmi sequenziali e' rappresentato dalla totale assenza di flessibilita'. Se ad esempio dobbiamo scrivere un programma sequenziale che esegue dieci volte un determinato calcolo, siamo costretti a scrivere dieci volte la stessa sequenza di istruzioni; appare molto piu' logico invece scrivere una sola volta la sequenza di istruzioni che esegue il calcolo, con la possibilita' di chiamare questa sequenza da qualsiasi punto del programma. In sostanza, la programmazione sequenziale rende inapplicabile il concetto di **sottoprogramma**; inoltre, l'impossibilita' di poter saltare da una parte all'altra del codice e in particolare, l'impossibilita' di saltare ad una istruzione precedente, impedisce la realizzazione di programmi capaci di eseguire cicli (iterazioni) o di prendere decisioni in base al risultato di una determinata operazione.
Tutte le CPU della famiglia **80x86** risolvono questi problemi mettendo a disposizione del programmatore una serie di istruzioni che permettono di effettuare il cosiddetto **trasferimento del controllo**; questa definizione e' legata al fatto che attraverso queste istruzioni e' possibile trasferire il controllo (saltare) ad un'altra istruzione che puo' trovarsi in qualunque altro punto del nostro programma. Per capire l'importanza di questo argomento basti pensare al fatto che i linguaggi di programmazione di alto livello sfruttano proprio queste istruzioni per implementare non solo il concetto di sottoprogramma, ma anche un'altra importante caratteristica chiamata **controllo del flusso**; attraverso il controllo del flusso e' possibile alterare il normale ordine di esecuzione delle istruzioni che formano un programma. Tra i vari strumenti che i linguaggi come il **C/C++** il **Pascal**, il **Basic**, etc, mettono a disposizione per il controllo del flusso si possono citare le istruzioni decisionali **IF**, **THEN**, **ELSE**, etc; altre istruzioni importanti sono quelle che ci permettono di effettuare le iterazioni (cicli **FOR**, cicli **WHILE-DO**, cicli **REPEAT-UNTIL**, etc).
Un programma che fa uso di sottoprogrammi e di istruzioni per il controllo del flusso puo' assumere una struttura piuttosto complessa che porta ad una inevitabile diminuzione delle prestazioni in termini di velocita' di esecuzione; al contrario, un programma sequenziale ha una struttura estremamente semplice che facilita enormemente il lavoro che la CPU deve svolgere per eseguire il programma stesso. In sostanza, se vogliamo spingere al massimo la velocita' dei nostri programmi dobbiamo puntare il piu' possibile sulla sequenzialita' delle varie istruzioni; se invece ci interessa scrivere programmi che eseguono compiti molto sofisticati, dobbiamo ricorrere necessariamente alle istruzioni per il trasferimento del controllo. E' chiaro che se si riesce a trovare un giusto equilibrio tra queste due esigenze, e' possibile scrivere programmi molto complessi e allo stesso tempo molto veloci; per raggiungere questo obbiettivo e' necessaria una conoscenza approfondita delle varie istruzioni per il trasferimento del controllo.

L'istruzione CALL.

Con il mnemonico **CALL** si indica l'istruzione **CALL procedure** (chiamata di un sottoprogramma); questa istruzione richiede un solo operando che viene utilizzato dalla CPU per determinare l'indirizzo al quale verra' trasferito il controllo. In sostanza, quando la CPU incontra il codice macchina dell'istruzione **CALL**, esegue un salto all'inizio di un gruppo di istruzioni che nel loro insieme formano un sottoprogramma; al termine del sottoprogramma, l'esecuzione riprende

dall'istruzione immediatamente successiva alla **CALL**.
Un sottoprogramma non e' altro che un blocco di istruzioni consecutive e contigue, distinte dalle istruzioni del programma principale; nel loro insieme queste istruzioni formano un vero e proprio "miniprogramma" che esegue un determinato compito. L'uso dei sottoprogrammi e' molto vantaggioso nel momento in cui il programma principale ha bisogno di ripetere numerose volte un determinato procedimento di calcolo (algoritmo); in questi casi, si puo' incorporare l'algoritmo in un apposito sottoprogramma che puo' essere chiamato dal programma principale ogni volta che se ne presenta la necessita'.
Tutti i linguaggi di programmazione di alto livello permettono l'uso dei sottoprogrammi che in genere vengono chiamati **funzioni** o **procedure**; queste denominazioni hanno lo scopo di richiamare alla mente il concetto di funzione matematica. In matematica il simbolo **z=f(x,y)** indica che il valore **z** (variabile dipendente) dipende dai due parametri **x** e **y** (variabili indipendenti); si dice anche che **z** e' funzione di **x** e di **y**. Ogni volta che vogliamo calcolare un nuovo valore di **z**, ci basta chiamare la funzione **f** passandole i due parametri **x** e **y**; la funzione esegue i calcoli e ci restituisce il risultato in **z** (valore di ritorno).
Nel linguaggio Assembly i sottoprogrammi vengono chiamati **procedure**; tutti i dettagli relativi alla creazione e alla gestione delle procedure vengono esposti in un apposito capitolo. In questo capitolo vengono analizzati invece i dettagli relativi al trasferimento del controllo dal programma principale ad una procedura e viceversa.
Come accade per le variabili, anche le procedure vengono identificate attraverso un nome simbolico; questo nome delimita l'inizio della procedura e il suo indirizzo coincide quindi con l'indirizzo della prima istruzione della procedura stessa. Tutto cio' implica che, come accade per le variabili, anche le procedure possono essere chiamate attraverso il loro nome o attraverso il loro indirizzo; nel primo caso si parla di chiamata diretta (**direct call**), mentre nel secondo caso si parla di chiamata indiretta (**indirect call**). Un'altro aspetto importante da tener presente e' dato dal fatto che la definizione della procedura puo' trovarsi nello stesso segmento di programma dal quale viene effettuata la chiamata, oppure puo' trovarsi in un segmento di programma differente; nel primo caso si parla di chiamata intrasegmento (**intrasegment call**), mentre nel secondo caso si parla di chiamata intersegmento (**intersegment call**).
Come e' stato spiegato in un precedente capitolo, quando si chiama una procedura si pone il problema di stabilire da quale indirizzo riprendera' l'esecuzione del programma principale una volta che la procedura ha terminato il suo lavoro; abbiamo visto che la soluzione piu' logica appare quella di far riprendere l'esecuzione dall'indirizzo dell'istruzione immediatamente successiva all'istruzione **CALL** (indirizzo di ritorno o **return address**). Per questo motivo, la CPU prima di chiamare una procedura, salva nello stack l'indirizzo di ritorno; al termine della procedura, l'indirizzo di ritorno viene estratto dallo stack e viene caricato in **CS:IP**.
Partiamo dal caso piu' semplice rappresentato dalla chiamata diretta intrasegmento (**direct call within segment**); in questo caso il codice macchina dell'istruzione **CALL** e' costituito dall'opcode **11101000b = E8h** seguito da un valore immediato (spiazzamento) a **16** bit che possiamo indicare con **DISP**. Lo spiazzamento **DISP** rappresenta un numero con segno a **16** bit che la CPU somma algebricamente all'offset dell'istruzione immediatamente successiva alla **CALL** per ottenere l'indirizzo della procedura da chiamare; vediamo un esempio pratico che chiarisce questa situazione. Supponiamo di aver definito all'offset **0070h** di un blocco codice chiamato **CODESEGM** una procedura rappresentata dal nome simbolico **ProcNear**; supponiamo ora che all'interno dello stesso blocco codice siano presenti le seguenti due istruzioni:

```
0015h  E8 0058h   call     ProcNear
0018h  8B D8h     mov      bx, ax
```

(alla sinistra di ciascuna istruzione vengono specificati l'offset e il codice macchina dell'istruzione stessa).

Come si puo' notare, l'istruzione **CALL** si trova all'offset **0015h** ed ha un codice macchina formato da **3** byte; l'istruzione successiva si trova di conseguenza all'offset **0018h** e rappresenta l'istruzione da cui riprendera' l'esecuzione al termine della procedura **ProcNear**. Vediamo allora quello che succede quando l'assembler arriva all'offset **0015h** del blocco codice del nostro programma e si imbatte nell'istruzione di chiamata della procedura **ProcNear**; trattandosi di una chiamata diretta intrasegmento, l'assembler genera il codice macchina **E8h** seguito dallo spiazzamento **DISP** a **16** bit. Per ricavare **DISP** l'assembler sottrae all'offset **0070h** della procedura, l'offset **0018h** dell'istruzione immediatamente successiva alla **CALL** ottenendo:
DISP = 0070h - 0018h = 0058h.
Quando la CPU incontra questo codice macchina, capisce che si tratta di una chiamata diretta intrasegmento (**E8h**); di conseguenza la CPU calcola l'indirizzo di **ProcNear** ottenendo:
0058h + 0018h = 0070h.
Il risultato cosi' ottenuto rappresenta un intero senza segno a **16** bit che permette alla CPU di saltare in qualunque altro punto di un segmento di programma; osserviamo infatti che con **16** bit possiamo esprimere tutti gli offset che vanno da **0000h** a **FFFFh**. Nel caso del nostro esempio, quest'offset viene caricato in **IP** per ottenere l'indirizzo **CS:IP** della prossima istruzione da eseguire; ovviamente si tratta dell'indirizzo della prima istruzione di **ProcNear**. Osserviamo che trattandosi di una chiamata intrasegmento, il contenuto di **CS** rimane inalterato; per questo motivo questo tipo di chiamata viene definita: chiamata **NEAR** di una procedura (near = vicino). In pratica, l'indirizzo della procedura da chiamare e' formato dal solo offset a **16** bit; il segmento in cui si trova la procedura non deve essere specificato in quanto e' implicitamente **CS**. Si tratta della stessa situazione che si verifica quando il nostro programma ha un solo segmento di dati referenziato da **DS**; per accedere a questi dati dobbiamo specificare solo il loro offset in quanto il segmento e' implicitamente **DS**.
Tornando a **ProcNear**, prima di chiamare questa procedura la CPU provvede a salvare l'indirizzo di ritorno nello stack; in conseguenza del fatto che si tratta di una chiamata di tipo **NEAR**, l'indirizzo di ritorno e' rappresentato dal solo offset **0018h** dell'istruzione immediatamente successiva alla **CALL**. A questo punto la CPU carica l'offset **0070h** in **IP** e salta all'indirizzo **CS:IP** che contiene la prima istruzione di **ProcNear**; al termine di **ProcNear** inizia il procedimento di ritorno al programma principale. Come viene spiegato piu' avanti, questo procedimento viene attivato dall'istruzione **RET** (return) che e' sempre l'ultima istruzione di una procedura; nel nostro caso si tratta di un ritorno di tipo **NEAR**. Il ritorno di tipo **NEAR** determina l'estrazione dallo stack di un valore a **16** bit che rappresenta l'offset della prossima istruzione da eseguire (indirizzo di ritorno); quest'offset viene caricato in **IP** e l'esecuzione salta a **CS:IP**. Nel nostro caso se tutto e' filato liscio, l'offset estratto dallo stack e' **0018h**, cioe' l'offset dell'istruzione immediatamente successiva alla **CALL**; come si puo' notare, sia nella fase di chiamata di **ProcNear** che nella fase di ritorno, il contenuto di **CS** e' rimasto inalterato.
Vediamo quello che succede quando la procedura **ProcNear** viene definita ad esempio all'offset **0002h** del blocco **CODESEGM**; in questo caso quando l'assembler incontra all'offset **0015h** l'istruzione:
`call ProcNear`, produce il codice macchina:
`E8 FFEAh`. Osserviamo infatti che l'offset **0018h** dell'istruzione che segue la **CALL** si trova **22** byte piu' avanti dell'offset **0002h** di **ProcNear**; la rappresentazione in complemento a due di **-22** e' **65514** che in esadecimale si scrive proprio **FFEAh**. Questo valore viene sommato algebricamente all'indirizzo **0018h** dell'istruzione immediatamente successiva alla **CALL** per ottenere l'indirizzo di **ProcNear**; la CPU quindi calcola:
FFEAh + 0018h = 10002h. Il riporto viene perso e si ottiene quindi **0002h**; questo valore viene caricato in **IP** e l'esecuzione del programma salta a **CS:IP**.

Passiamo ora alla chiamata indiretta intrasegmento di una procedura (**indirect call within segment**); questa situazione si verifica quando l'operando dell'istruzione **CALL** contiene l'indirizzo della procedura da chiamare. Il codice macchina di questa istruzione e' formato dai due opcodes **11111111b = FFh** e **mod_010_r/m**; dalla struttura dell'opcode secondario si intuisce che per contenere l'indirizzo della procedura da chiamare possiamo utilizzare una variabile a **16** bit o un registro generale a **16** bit. Come al solito, trattandosi di una chiamata intrasegmento, l'indirizzo a cui saltare e' formato dal solo offset a **16** bit (indirizzo **NEAR**); se ad esempio vogliamo chiamare indirettamente la procedura **ProcNear**, la prima cosa da fare consiste nel definire un'apposita variabile a **16** bit destinata a contenere l'indirizzo della procedura stessa. Possiamo scrivere ad esempio:

```
ProcNearAddr dw offset ProcNear;
```

come al solito e' fondamentale la presenza della varie direttive **ASSUME** che consentono all'assembler di determinare gli indirizzi in modo corretto. Supponiamo che **ProcNear** si trovi all'offset **0070h** del blocco **CODESEGM**, e che **ProcNearAddr** si trovi all'offset **0002h** del blocco **DATASEGM**; possiamo dire quindi che la variabile **ProcNearAddr** conterra' il valore **0070h**. Consideriamo ora le seguenti due istruzioni presenti nel blocco **CODESEGM**; come al solito alla sinistra di ciascuna istruzione e' presente il relativo offset e il relativo codice macchina.

```
0015h  FF 16 0002h   call     ProcNearAddr
0019h  8B D8h        mov      bx, ax
```

In questo caso l'istruzione **CALL** si trova all'offset **0015h** ed ha un codice macchina formato da **4** byte; l'istruzione successiva si trova di conseguenza all'offset **0019h** e rappresenta l'istruzione da cui riprendera' l'esecuzione al termine della procedura **ProcNear**. Quando l'assembler esamina l'istruzione **CALL**, capisce che si tratta di una chiamata indiretta e genera quindi il primo opcode **FFh**; per indicare che la chiamata e' intrasegmento l'assembler utilizza il campo **reg** del secondo opcode ponendo **reg=010**. L'indirizzo di **ProcNear** e' contenuto nella locazione di memoria **ProcNearAddr** per cui l'assembler pone **mod_r/m = 00_110** per indicare alla CPU che l'accesso in memoria avviene tramite l'offset **0002h** di **ProcNearAddr**; il secondo opcode vale quindi **16h** ed e' seguito dall'offset **0002h** di **ProcNearAddr**. In fase di esecuzione il codice macchina **FFh** e il campo **reg=010** indicano alla CPU la presenza di una istruzione di chiamata indiretta intrasegmento ad una procedura; la CPU salva nello stack l'indirizzo di ritorno **0019h** e determina l'indirizzo a cui saltare per raggiungere la procedura stessa. Questa volta l'indirizzo e' contenuto direttamente nella variabile **ProcNearAddr** che si trova all'offset **0002h** del blocco **DATASEGM**; la CPU accede all'indirizzo **DS:0002h**, legge **16** bit contenenti il valore **0070h**, carica questo valore in **IP** e salta a **CS:IP**. Al termine della procedura viene attivato un ritorno di tipo **NEAR**; in questa fase la CPU estrae il valore **0019h** dallo stack, lo carica in **IP** e salta a **CS:IP**.
In alternativa al metodo appena illustrato, possiamo anche utilizzare un registro generale a **16** bit scrivendo ad esempio le istruzioni:

```
0015h  BA 0070h   mov      dx, offset ProcNear
0018h  FF D2h     call     dx
```

Non e' necessario utilizzare un registro puntatore in quanto lo scopo del registro e' solo quello di contenere un indirizzo che la CPU carichera' in **IP**; come possiamo notare, la prima istruzione carica in **DX** il valore **0070h** che e' l'offset della procedura **ProcNear**. Per quanto riguarda la seconda istruzione, il primo opcode e' come al solito **FFh** che indica una chiamata indiretta; nel secondo opcode il campo **mod** vale **11b** per indicare che il campo **r/m** contiene il codice di un registro. Trattandosi del registro **DX** otteniamo **r/m=010b** e quindi il secondo opcode vale proprio **D2h**; quando la CPU incontra questo codice macchina procede nel modo gia' descritto in precedenza nel caso di **ProcNearAddr**.

La situazione diventa piu' delicata nel caso in cui la definizione della procedura si trovi in un segmento di programma diverso da quello che contiene la chiamata alla procedura stessa; in un caso del genere, per saltare alla procedura la CPU deve modificare sia **IP** che **CS**. Partiamo dalla chiamata diretta intersegmento (**direct call intersegment**); in questo caso il codice macchina dell'istruzione **CALL** e' costituito dall'opcode **10011010b = 9Ah** seguito da un valore immediato a **16+16=32** bit che rappresenta l'indirizzo della procedura in formato **SEG:OFFSET**. Supponiamo di aver definito all'offset **0003h** di un blocco codice chiamato **CODESEGM2** una procedura rappresentata dal nome simbolico **ProcFar**; supponiamo ora che all'interno di un diverso blocco codice chiamato **CODESEGM** siano presenti le seguenti due istruzioni:

```
0015h  A9 00000003h  call     ProcFar
001Ah  8B D8h        mov      bx, ax
```

L'istruzione **CALL** si trova all'offset **0015h** ed ha un codice macchina formato da **5** byte; l'istruzione successiva si trova di conseguenza all'offset **001Ah** e rappresenta l'istruzione da cui riprendera' l'esecuzione al termine della procedura **ProcFar**. Vediamo allora quello che succede quando l'assembler arriva all'offset **0015h** del blocco codice del nostro programma e si imbatte nell'istruzione di chiamata della procedura **ProcFar**; trattandosi di una chiamata diretta intersegmento, l'assembler genera il codice macchina **A9h** seguito dall'indirizzo **SEG:OFFSET** di **ProcFar**. Nel rispetto della convenzione **LITTLE-ENDIAN**, la componente **OFFSET** deve occupare la word meno significativa dell'indirizzo, mentre la componente **SEG** deve occupare la word piu' significativa; nel nostro caso vediamo che subito dopo l'opcode **A9h** e' presente il valore immediato a **32** bit **00000003h** che deve essere interpretato come coppia **0000h:0003h**. La componente **OFFSET** occupa la word meno significativa e vale **0003h**; la componente **SEG** occupa la word piu' significativa e vale simbolicamente **0000h** in quanto il suo vero valore verra' assegnato dal **SO** al momento del caricamento del programma in memoria.
Quando la CPU incontra questo codice macchina, capisce che si tratta di una chiamata diretta intersegmento (**A9h**); questo significa che per poter saltare alla procedura la CPU deve modificare non solo **IP** ma anche **CS**. Questo tipo di chiamata viene definita: chiamata **FAR** di una procedura (far = lontano); la conseguenza di tutto cio' e' che prima di chiamare **ProcFar** la CPU salva nello stack un indirizzo di ritorno formato non solo dall'offset **001Ah** dell'istruzione immediatamente successiva alla **CALL**, ma anche dal contenuto corrente di **CS** (che referenzia **CODESEGM**). Seguendo la convenzione **LITTLE-ENDIAN** la CPU inserisce nello stack prima il contenuto corrente di **CS** e poi l'offset **001Ah**; in questo modo nello stack la componente **OFFSET** precede immediatamente la componente **SEG**. A questo punto, la CPU legge dal codice macchina l'indirizzo **SEG:OFFSET** di **ProcFar** lo carica in **CS:IP** e salta a **CS:IP** che e' l'indirizzo della prima istruzione di **ProcFar**; al termine di **ProcFar** inizia il procedimento di ritorno al programma principale. Questa volta si tratta di un ritorno di tipo **FAR** che determina l'estrazione dallo stack di due valori a **16** bit; il primo valore viene caricato in **IP**, mentre il secondo valore viene caricato in **CS**. Se tutto e' filato liscio, il valore caricato in **IP** e' **001Ah**, mentre il valore caricato in **CS** e' il paragrafo di memoria da cui parte **CODESEGM**; l'esecuzione riprende quindi da **CODESEGM:001Ah** che e' l'indirizzo dell'istruzione immediatamente successiva alla **CALL**.

Passiamo infine alla chiamata indiretta intersegmento (**indirect call intersegment**); in questo caso il codice macchina dell'istruzione **CALL** e' costituito dai due opcodes **11111111b = FFh** e **mod_011_r/m**. Anche nella chiamata indiretta intersegmento, l'istruzione **CALL** utilizza un operando contenente l'indirizzo completo **SEG:OFFSET** della procedura da chiamare; questa volta pero' l'indirizzo deve essere specificato dal programmatore. Supponiamo come al solito di avere una procedura **ProcFar** definita all'offset **0003h** di un segmento di codice chiamato **CODESEGM2**; vogliamo chiamare indirettamente questa procedura da un'altro segmento di codice chiamato

CODESEGM. Prima di tutto creiamo una variabile destinata a contenere l'indirizzo completo di **ProcFar**; possiamo scrivere ad esempio:
`ProcFarAddr dd 0.`
A questo punto, nel blocco **CODESEGM** dobbiamo procedere all'inizializzazione della variabile **ProcFarAddr** che verra' utilizzata per la chiamata indiretta intersegmento; supponendo che **ProcFarAddr** si trovi all'offset **000Ah** del blocco **DATASEGM**, otteniamo le seguenti istruzioni:

```
0050h  C7 06 000A 0003h    mov      word ptr ProcFarAddr[0], offset ProcFar
0056h  C7 06 000C 0000h    mov      word ptr ProcFarAddr[2], seg ProcFar
005Ch  FF 1E 000Ah         call     ProcFarAddr
0060h  8B D8h              mov      bx, ax
```

L'aspetto fondamentale da ricordare e' che l'indirizzo di **ProcFar** deve essere caricato nella variabile **ProcFarAddr** a **32** bit nel rispetto della convenzione **LITTLE-ENDIAN**; questo significa che la componente **OFFSET** deve essere caricata nella word meno significativa di **ProcFarAddr**, mentre la componente **SEG** deve essere caricata nella word piu' significativa. La prima istruzione dell'esempio si trova all'offset **0050h** e carica l'offset **0003h** di **ProcFar** nei primi **16** bit di **ProcFarAddr** che partono dall'offset **000Ah** di **DATASEGM**; si tratta di un trasferimento dati da **imm16** a **mem16**. La seconda istruzione si trova all'offset **0056h** e carica la componente **SEG** di **ProcFar** nei **16** bit piu' significativi di **ProcFarAddr** che partono dall'offset **000Ch** di **DATASEGM**; come al solito l'assembler usa un valore simbolico **0000h** in quanto il vero contenuto di **CODESEGM2** verra' assegnato dal **SO** al momento di caricare il programma in memoria. A questo punto arriviamo all'istruzione di chiamata che si trova all'offset **005Ch** del blocco **CODESEGM**; l'assembler osservando che l'operando di **CALL** e' una variabile a **32** bit capisce che si tratta di una chiamata indiretta e genera quindi il primo opcode **FFh**. Per il secondo opcode l'assembler pone **reg=011** per indicare che la chiamata e' intersegmento; inoltre **mod_r/m=00_110** per indicare che l'indirizzo da chiamare si trova in una variabile. Il secondo opcode vale quindi **1Eh** ed e' seguito dall'offset **000Ah** di **ProcFarAddr**; questo codice macchina occupa **4** byte per cui l'istruzione successiva si trova all'offset **0060h** ed e' l'istruzione da cui riprendera' l'esecuzione al termine della procedura.
In fase di esecuzione il codice macchina **FFh** e il campo **reg=011** indicano alla CPU la presenza di una istruzione di chiamata indiretta intersegmento ad una procedura; la CPU salva nello stack l'indirizzo di ritorno formato dal contenuto corrente di **CS** (**CODESEGM**) e dall'offset **0060h**. Per determinare l'indirizzo a cui saltare la CPU accede alla locazione di memoria **DS:000Ah** legge **32** bit contenenti la coppia **CODESEGM2:0003h**, carica questa coppia in **CS:IP** e salta a **CS:IP**; al termine della procedura viene attivato un ritorno di tipo **FAR** con la CPU che estrae dallo stack la coppia **CODESEGM:0060h**, la carica in **CS:IP** e salta a **CS:IP**.
In analogia all'esempio relativo alla chiamata indiretta intrasegmento, si potrebbe pensare di utilizzare un registro a **32** bit per la chiamata indiretta intersegmento; questo pero' non e' possibile in quanto dobbiamo tener presente che stiamo operando in modalita' reale con segmenti di programma aventi attributo **SIZE** di tipo **USE16**. Come gia' sappiamo, in questa modalita' la CPU lavora solamente con indirizzi formati da una componente **SEG** a **16** bit e una componente **OFFSET** a **16** bit; caricando ad esempio nel registro **EBX** l'indirizzo **SEG:OFFSET** di una procedura chiamata **ProcFar** e scrivendo l'istruzione:
`66 FF D3h call ebx`, non otteniamo il risultato desiderato. Infatti quando la CPU esegue questa istruzione, pur incontrando l'**Operand Size Prefix = 66h**, utilizza solo i **16** bit meno significativi di **EBX** e cerca di effettuare una salto **NEAR** ad una procedura che si trova in un'altro segmento di programma; come si puo' notare infatti, l'assembler ha generato il primo opcode **FFh** per la chiamata indiretta, e ha posto pero' **reg=010** per la chiamata intrasegmento (**mod=11** e **r/m=EBX=011**). Si tratta chiaramente di una situazione di errore che generalmente manda in crash il nostro programma.

In base a tutte le considerazioni svolte a proposito dell'istruzione **CALL**, si capisce subito che la chiamata **FAR** e' piu' lenta della chiamata **NEAR**; cio' e' dovuto principalmente al fatto che nel caso di chiamata **FAR** la CPU deve maneggiare indirizzi **FAR** formati da coppie **SEG:OFFSET**, piu' pesanti da gestire rispetto agli indirizzi **NEAR** formati dalla sola componente **OFFSET**. Un'altro aspetto da considerare riguarda il fatto che il ricorso massiccio ai sottoprogrammi, comporta una conseguente diminuzione delle prestazioni del nostro programma a causa dei continui salti da un punto all'altro del segmento di codice; in particolare, la chiamata di una procedura richiede un certo intervallo di tempo necessario alla CPU per salvare nello stack l'indirizzo di ritorno, e analogamente, l'uscita da una procedura richiede un certo intervallo di tempo necessario alla CPU per estrarre dallo stack lo stesso indirizzo di ritorno.
Un'ultima considerazione riguarda il fatto che tutti i concetti appena esposti sull'istruzione **CALL** sono relativi alla modalita' reale **8086**; in questa modalita', l'esecuzione dell'istruzione **CALL** non produce nessun effetto sui bit del **Flags Register**.

L'istruzione RET.

Con il mnemonico **RET** si indica l'istruzione **RETurn from procedure** (ritorno da un sottoprogramma); come si puo' facilmente intuire, questa istruzione lavora in coppia con **CALL** ed e' sempre l'ultima istruzione di una procedura. Quando la CPU incontra questa istruzione, estrae un valore dallo stack (indirizzo di ritono), lo carica in **CS:IP** e salta a **CS:IP**; in assenza di errori, l'indirizzo di ritorno coincide con l'indirizzo dell'istruzione immediatamente successiva alla **CALL**.
Abbiamo visto che la chiamata di una procedura puo' essere intrasegmento o intersegmento; in corrispondenza di questi due tipi di chiamata, anche il ritorno da una procedura puo' essere intrasegmento o intersegmento (il concetto di ritorno diretto o indiretto non ha invece nessun significato).
Partiamo dal caso del ritorno intrasegmento (**return within segment** o **near return**); in questo caso il codice macchina dell'istruzione **RET** e' formato dal solo opcode **11000011b = C3h**. Quando la CPU incontra questo codice macchina, estrae dallo stack un valore a **16** bit, lo carica in **IP** e salta a **CS:IP**; a meno di errori, il valore estratto dallo stack coincide con l'offset dell'istruzione immediatamente successiva a quella che ha chiamato la procedura. Naturalmente il ritorno intrasegmento da una procedura e' associato alla chiamata intrasegmento della procedura stessa; si noti che in questo caso il contenuto di **CS** rimane inalterato.
Passiamo al caso del ritorno intersegmento (**intersegment return** o **far return**); in questo caso il codice macchina dell'istruzione **RET** e' formato dal solo opcode **11001011b = CBh**. Quando la CPU incontra questo codice macchina, estrae dallo stack un valore a **16** bit e lo carica in **IP**, poi estrae dallo stack un'altro valore a **16** bit e lo carica in **CS**, infine salta a **CS:IP**; a meno di errori, la coppia di valori estratta dallo stack coincide con l'indirizzo **SEG:OFFSET** dell'istruzione immediatamente successiva a quella che ha chiamato la procedura. Naturalmente il ritorno intersegmento da una procedura e' associato alla chiamata intersegmento della procedura stessa; si noti che in questo caso il contenuto di **CS** viene modificato in quanto la procedura si trova in un segmento di programma diverso da quello contenente la chiamata alla procedura stessa. Per rendere esplicito il fatto che si tratta di un **far return**, il programmatore puo' anche utilizzare l'istruzione **RETF** (far return); in questo caso l'assembler fidandosi del programmatore inserisce direttamente il codice macchina **CBh**.

L'istruzione **RET** si presenta anche in una forma particolare che prevede la presenza di un operando immediato a **16** bit; questo operando contiene un valore in byte che la CPU somma al registro **SP** (Stack Pointer) prima di saltare all'indirizzo di ritorno contenuto in **CS:IP**. Questo aspetto assume una notevole importanza relativamente al metodo che si utilizza per il passaggio di eventuali parametri alle procedure; dal punto di vista dell'Assembly, il metodo piu' veloce per passare i parametri alle procedure consiste nell'utilizzare i registri della CPU. Molti linguaggi di

programmazione di alto livello, utilizzano invece lo stack; questo significa che al termine della procedura si presenta il problema della "pulizia" dello stack, cioe' della rimozione dei parametri dallo stack. Rimuovere i parametri dallo stack significa in sostanza incrementare il registro **SP** (Stack Pointer) in modo da recuperare lo spazio precedentemente occupato nello stack dai parametri stessi; uno dei possibili metodi per la pulizia dello stack consiste proprio nell'utilizzare l'istruzione **RET** seguita da un valore immediato a **16** bit che indica alla CPU di quanti byte bisogna incrementare **SP** al termine della procedura. Questa tecnica viene utilizzata ad esempio dai compilatori **Pascal**; per questo motivo si parla anche di convenzione **Pascal** per la pulizia dello stack. Supponiamo ad esempio di avere una procedura **Pascal** chiamata **ProcPascal** che richiede tre parametri di tipo **Integer** (intero con segno a **16** bit); quando il compilatore **Pascal** incontra la seguente chiamata:
`ProcPascal(a, b, c);`,
la converte in una sequenza di codici macchina corrispondenti alle seguenti istruzioni Assembly:

```
push     a            ; primo parametro
push     b            ; secondo parametro
push     c            ; terzo parametro
call     ProcPascal   ; chiamata della procedura
```

Siccome vengono inseriti nello stack **3** parametri da **2** byte ciascuno, abbiamo un totale di **6** byte; al termine della procedura **ProcPascal**, il compilatore **Pascal** inserisce l'istruzione:
`RET 6` (si puo' anche scrivere `RET (6)`).
Analizziamo in dettaglio la situazione supponendo che inizialmente **SP** contenga il valore **0F80h** e che la chiamata sia di tipo **NEAR**; per ogni parametro inserito nello stack **SP** viene decrementato di **2**, per cui dopo l'inserimento di **a**, **b** e **c** si ottiene **SP=0F7Ah**. Subito dopo troviamo la chiamata **NEAR** che comporta l'inserimento nello stack dell'indirizzo di ritorno a **16** bit; di conseguenza **SP** viene decrementato di **2** e si ottiene **SP=0F78h**. A questo punto la CPU carica in **IP** l'offset di **ProcPascal** e salta a **CS:IP**; al termine della procedura la CPU incontra l'istruzione **RET** seguita dal valore immediato **6**. In base a questa istruzione, prima di tutto la CPU estrae dallo stack l'indirizzo di ritorno a **16** bit e incrementa **SP** di **2** ottenendo **SP=0F7Ah**; successivamente la CPU somma **6** byte a **SP** ottenendo **SP=0F80h**. L'ultima fase consiste nel caricare in **IP** l'indirizzo di ritorno per poi saltare a **CS:IP**; come si puo' notare, al termine della procedura **SP** contiene **0F80h**, cioe' lo stesso valore che aveva prima della chiamata di **ProcPascal**. Naturalmente tutte queste considerazioni presuppongono che la procedura **ProcPascal** non commetta errori nella gestione dello stack; in caso contrario al termine della procedura, **SP** si viene a trovare nel posto sbagliato. Di conseguenza la CPU estrae dallo stack un indirizzo di ritorno privo di senso, lo carica in **CS:IP**, salta nel vuoto e il programma va in crash.

Le considerazioni appena esposte, vengono illustrate in dettaglio in appositi capitoli dedicati alle procedure e all'interfacciamento dell'Assembly con i linguaggi di alto livello; anche questa forma particolare dell'istruzione **RET** prevede naturalmente i due casi di ritorno intrasegmento e ritorno intersegmento. Per il ritorno intrasegmento il codice macchina di **RET** e' formato dall'opcode **11000010b = C2h** seguito da un valore **imm16**; per il ritorno intersegmento il codice macchina di **RET** e' formato dall'opcode **11001010b = CAh** seguito da un valore **imm16**. In modalita' reale **8086** l'esecuzione dell'istruzione **RET** non altera nessun campo del **Flags Register**.

Le istruzioni INT e INTO.

Con il mnemonico **INT** si indica l'istruzione **call to INTerrupt procedure** (chiamata di un gestore di interruzione); attraverso questa istruzione i programmi possono interfacciarsi con i servizi offerti dal **BIOS**, dal **DOS**, dai **Device Drivers**, etc.
Quando si parla di **Personal Computer** o **PC**, si fa riferimento ad una categoria di computers dotati di piattaforme hardware compatibili tra loro; questa compatibilita' e' basata sul rigoroso rispetto di una serie di standard, molti dei quali vennero introdotti dalla **IBM** negli anni **80** (per questo motivo un tempo si parlava anche di **PC IBM compatibili**). Una delle convenzioni piu' importanti che i **PC** devono rispettare, riguarda il fatto che i primi **1024** byte della memoria **RAM**, cioe' tutti i byte che occupano gli indirizzi fisici compresi tra **00000h** e **00400h**, devono essere tassativamente riservati ai cosiddetti **vettori di interruzione** (interrupt vectors); con il termine **vettore di interruzione** si indica un indirizzo **FAR**, formato quindi da una coppia **SEG:OFFSET**, che punta ad una determinata procedura chiamata in genere **interrupt service routine** o **ISR**. Ciascun vettore di interruzione, essendo composto da una coppia **SEG:OFFSET** da **16+16** bit, occupa **4** byte; complessivamente quindi in **1024** byte possono trovare posto **1024/4=256** vettori di interruzione che vengono infatti indicati con i numeri da **0** a **255**. Molti vettori di interruzione sono riservati al **BIOS** e al **DOS**; altri vettori di interruzione sono a disposizione dei programmatori che possono usarli per chiamare le **ISR** personali. Appena si accende il computer, il controllo viene preso dal **BIOS** che tra le altre cose, provvede anche ad installare in memoria una serie di **ISR** destinate alla gestione a basso livello dell'hardware; per ciascuna **ISR** il **BIOS** installa il relativo vettore di interruzione. Successivamente il **BIOS** cede il controllo al **SO** che a sua volta provvede ad installare in memoria altre **ISR** con i relativi vettori di interruzione; nel caso del **DOS**, vengono installate una serie di **ISR** che hanno lo scopo di offrire ai programmi vari servizi come la gestione dei files su disco, la gestione della memoria, etc. Altre **ISR** vengono installate inoltre dai **Device Drivers** (piloti di dispositivo); attraverso queste **ISR** e' possibile gestire diverse periferiche collegate al computer.
Se vogliamo utilizzare queste **ISR**, dobbiamo servirci dell'istruzione **INT**; questa istruzione richiede un operando di tipo **imm8** che contiene un numero compreso tra **0** e **255** attraverso il quale possiamo specificare il vettore di interruzione desiderato. Il codice macchina dell'istruzione **INT** e' formato dall'opcode **11001101b = CDh** seguito dal valore immediato **imm8**; quando la CPU incontra questo codice macchina attiva una fase molto simile alla chiamata **FAR** di una normale procedura. Vediamo in dettaglio quello che succede quando la CPU incontra il codice macchina dell'istruzione:
`INT n` (dove **n** indica un generico vettore di interruzione); la CPU inserisce in successione nello stack il contenuto a **16** bit del **Flags Register**, il contenuto corrente di **CS** e l'offset dell'istruzione immediatamente successiva alla **INT**. Dopo questa fase la CPU legge la coppia **SEG:OFFSET** corrispondente al vettore di interruzione in posizione **n**, carica questa coppia in **CS:IP** e salta a **CS:IP**, cioe' alla prima istruzione della **ISR**; come viene spiegato piu' avanti, al termine della **ISR**, l'apposita istruzione **IRET** (interrupt return) attiva il procedimento di ritorno al chiamante. Come si puo' notare, l'istruzione **INT** prima di chiamare l'**ISR** salva nello stack non solo l'indirizzo di ritorno ma anche il registro **FLAGS**; questo aspetto e' molto importante in quanto alcune **ISR** vengono chiamate direttamente dall'hardware mentre il nostro programma e' in fase di esecuzione. In un caso del genere, la CPU interrompe immediatamente il nostro programma ed effettua la chiamata dell'**ISR**; e' fondamentale quindi che in questa fase, lo stato del nostro programma venga preservato e questo significa che la chiamata dell'**ISR** deve preservare il contenuto di tutti i registri della **CPU** che vengono eventualmente utilizzati dalla **ISR** stessa. La CPU provvede a preservare il contenuto di **FLAGS**, mentre il compito di preservare il contenuto degli altri registri spetta alla **ISR**; queste considerazioni naturalmente valgono anche per le **ISR** scritte dal programmatore.
Tutti gli aspetti relativi ai vettori di interruzione vengono illustrati in dettaglio nella sezione **Assembly Avanzato**; a titolo di curiosita', vediamo un esempio relativo al vettore di interruzione n.

21h. Questo vettore di interruzione viene installato dal **DOS** e fornisce una serie di importanti servizi generali (gestione files, memoria, stampante, etc); generalmente il servizio desiderato viene specificato attraverso il registro **AH**, mentre gli eventuali parametri vengono passati alla **ISR** attraverso gli altri registri della CPU. Nel nostro caso richiediamo il servizio n. **09h** chiamato **Display String**; questo servizio mostra una stringa sullo schermo a partire dalla posizione corrente del prompt del **DOS**. La stringa deve essere puntata da **DS:DX** e deve terminare con il codice ASCII del simbolo **'$'** (dollaro); prima di tutto creiamo la stringa con la definizione:
`dosString db 'Chiamata del servizio Display String (INT 21h)', '$'.`
A questo punto possiamo procedere con la chiamata del servizio **Display String** attraverso le seguenti istruzioni:

```
mov      dx, offset dosString    ; ds:dx -> dosString
mov      ah, 09h                 ; servizio Display String
int      21h                     ; mostra la stringa
```

Questo esempio presuppone che la stringa sia stata definita in un blocco dati referenziato da **DS**; in questo caso **DS** contiene gia' il segmento di appartenenza della stringa. Nei precedenti capitoli abbiamo fatto conoscenza con il servizio n. **4Ch** dell'**INT 21h**; questo servizio ci permette di terminare il nostro programma restituendo il controllo al **DOS**.

Nel caso particolare del vettore di interruzione n. **03h**, l'assembler genera il codice macchina formato dal solo opcode **11001100b = CCh** eliminando l'operando **imm8**; il vettore di interruzione n. **03h** viene chiamato **Trap to Debugger** o **Breakpoint** e viene usato da appositi software chiamati appunto **debuggers**. I debuggers vengono impiegati per la ricerca di errori (bugs) nei programmi; l'esecuzione dell'**INT 03h** provoca la chiamata di un'apposita **ISR** predefinita che in genere contiene esclusivamente l'istruzione **IRET**. Naturalmente il debbugger installa una propria **ISR** che intercetta l'**INT 03h**; ogni volta che l'**ISR** viene chiamata, il debugger e' in grado di analizzare lo stato del nostro programma in quel preciso istante. Un metodo particolare per effettuare il debugging di un programma consiste nel porre a **1** il campo **TF** (Trap Flag) del **Flags Register**; in questo caso la CPU ogni volta che esegue un'istruzione del nostro programma genera automaticamente un'**INT 01h** chiamata **single step** (passo singolo). Un debugger che intercetta l'**INT 01h** e' in grado quindi di esaminare il nostro programma istruzione per istruzione; questo tipo di analisi viene chiamata **esecuzione passo passo**.

Un'altro caso particolare e' rappresentato dall'**INT 04h** chiamata **Interrupt on Overflow**; questa interruzione puo' essere utilizzata dal programmatore per gestire i casi di overflow che possono verificarsi nel corso di operazioni logico aritmetiche. Quando la CPU esegue l'**INT 04h**, controlla innanzi tutto il contenuto di **OF** (Overflow Flag); se **OF=0** l'esecuzione prosegue con l'istruzione successiva alla **INT**, mentre se **OF=1** la CPU chiama un'apposita **ISR** predefinita che in genere contiene esclusivamente l'istruzione **IRET**. Il programmatore puo' installare una propria **ISR** contenente il codice necessario per la gestione dei casi di overflow; questo aspetto e' molto importante nel caso in cui l'overflow provocato da una operazione deve essere considerato come una condizione di errore. Al posto dell'istruzione **INT 04h** e' possibile utilizzare l'istruzione equivalente **INTO**; in questo caso l'assembler genera un codice macchina formato dal solo opcode **11001110b = CEh**. Inserendo l'istruzione **INTO** subito dopo un'operazione logico aritmetica, si ottiene la chiamata dell'**INT 04h** solo se l'operazione stessa ha provocato un'overflow; nella sezione **Assembly Avanzato** vengono illustrati diversi esempi relativi ai metodi di installazione e gestione di una **ISR**.

L'esecuzione dell'istruzione **INT** modifica i campi **TF** (Trap Flag) e **IF** (Interrupt Enable Flag) del **Flags Register**; entrambi questi campi vengono posti a **0**. Il campo **TF** viene posto a **0** per evitare che l'esecuzione dell'**ISR** venga "disturbata" dalla generazione di una **INT 01h** per ogni istruzione della **ISR** stessa; il campo **IF** viene posto a **0** per evitare che nel corso dell'esecuzione dell'**ISR** sopraggiungano altre richieste di interruzione. Come gia' sappiamo, le richieste di interruzione che possono essere bloccate ponendo **IF=0** vengono definite **interruzioni mascherabili** (maskable interrupts); viceversa, le richieste di interruzione che non vengono influenzate dal contenuto di **IF** vengono definite **interruzioni non mascherabili** o **NMI** (Non Maskable Interrupts). La **NMI** piu' importante e' la **INT 02h** che viene generata direttamente dall'hardware del computer per segnalare problemi hardware di estrema gravita'; naturalmente ci auguriamo tutti che alla CPU del nostro computer non arrivi mai la **INT 02h**. Le interruzioni che vengono generate direttamente dall'hardware del computer vengono definite appunto **interruzioni hardware**; queste interruzioni vengono anche definite **asincrone** in quanto possono arrivare alla CPU in qualsiasi momento senza nessuna sincronizzazione con il programma in esecuzione. Le interruzioni generate dai programmi con l'istruzione **INT** vengono invece definite **interruzioni software**; ovviamente in questo caso si tratta di interruzioni di tipo **sincrono**.

L'istruzione IRET.

Con il mnemonico **IRET** viene indicata l'istruzione **Interrupt RETurn** (ritorno da un gestore di interruzione); questa istruzione attiva la fase di uscita da una **ISR** trasferendo il controllo all'istruzione immediatamente successiva alla **INT** che ha chiamato la **ISR** stessa. Abbiamo visto in precedenza che l'istruzione **RET** lavora in coppia con l'istruzione **CALL**; analogamente, l'istruzione **IRET** lavora in coppia con l'istruzione **INT**. Il codice macchina dell'istruzione **IRET** e' formato dal solo opcode **11001111b = CFh**; non e' previsto nessun operando immediato in quanto gli eventuali parametri richiesti da una **ISR** devono essere passati esclusivamente attraverso i registri della CPU.
L'esecuzione dell'istruzione **IRET** attiva una fase molto simile al ritorno **FAR** da una procedura; ovviamente **IRET** svolge un lavoro opposto a quello svolto da **INT**. Quando la CPU incontra il codice macchina di **IRET**, estrae dallo stack un valore a **16** bit e lo carica in **IP**, poi estrae un secondo valore a **16** bit e lo carica in **CS**, quindi estrae un terzo valore a **16** bit e lo carica in **FLAGS**, infine salta a **CS:IP**; a meno di errori, l'indirizzo caricato in **CS:IP** coincide con l'indirizzo dell'istruzione immediatamente successiva alla **INT** che aveva chiamato la **ISR**.
Ovviamente l'esecuzione di **IRET** altera tutti i bit del registro **FLAGS**; infatti la fase di uscita da una **ISR** comporta anche il ripristino del vecchio contenuto del registro dei flags, precedentemente salvato nello stack dall'istruzione **INT**. Normalmente quando si accende il computer, il **Flags Register** viene inizializzato con **IF=1** (interruzioni non mascherabili abilitate); questo aspetto e' fondamentale per il corretto funzionamento del computer. Per fare un esempio pratico, ponendo **IF=0** si impedisce l'aggiornamento dell'orologio del **BIOS**; e' importantissimo quindi evitare di tenere per lungo tempo **IF=0**. Le istruzioni **INT** e **IRET** rappresentano un esempio pratico di come ci si debba comportare nell'eventualita' di dover azzerare temporaneamente **IF**; l'istruzione **INT** pone **IF=0** e chiama una **ISR** che svolge il suo lavoro nel piu' breve tempo possibile in modo che la conseguente istruzione **IRET** riporti **IF** a **1**.

Le istruzioni LOOP, LOOPZ/LOOPE, LOOPNZ/LOOPNE.

Con il mnemonico **LOOP** si indica l'istruzione **LOOP control with CX counter** (controllo di una iterazione attraverso il registro **CX**); questa istruzione permette di ripetere (iterare) l'esecuzione di un blocco di istruzioni per un numero di volte indicato dal contenuto del registro **CX**. La struttura classica di un loop e' formata in genere dalle seguenti parti:

1) un blocco di istruzioni di inizializzazione;
2) una etichetta che delimita l'inizio del loop;
3) un blocco di istruzioni che formano il corpo del loop;
4) l'istruzione **LOOP** che controlla l'iterazione.

Le istruzioni di inizializzazione comprendono in particolare il caricamento del **contatore** nel registro **CX**; diverse istruzioni Assembly utilizzano implicitamente **CX** come registro contatore e questo spiega il nome **Counter Register** che viene dato a **CX**. Il codice macchina dell'istruzione **LOOP** e' formato dall'opcode **11100010b = E2h** seguito da un valore immediato **DISP** a **8** bit che viene interpretato dalla CPU come numero intero con segno compreso quindi tra **-128** e **+127**. Quando la CPU incontra il codice macchina dell'istruzione **LOOP**, decrementa **CX** di **1** e compie una scelta sulla base del nuovo contenuto di **CX**; se **CX=0** il loop viene considerato concluso e l'esecuzione prosegue con l'istruzione immediatamente successiva a **LOOP**. Se invece **CX** e' diverso da **0**, la CPU somma algebricamente **DISP** all'offset dell'istruzione immediatamente successiva a **LOOP** e ottiene l'offset dell'etichetta che delimita l'inizio del loop; quest'offset viene caricato in **IP** e l'esecuzione salta a **CS:IP** determinando una nuova iterazione del blocco di istruzioni che formano il corpo del loop. Vediamo un esempio pratico che chiarisce questi concetti; supponiamo di aver definito all'offset **000Eh** del blocco dati del nostro programma il seguente vettore:

```
vectorWord dw 100 dup (0).
```

Si tratta di un vettore formato da **100** elementi a **16** bit; ciascun elemento del vettore viene inizializzato dal valore **0**. Supponiamo ora di voler caricare il valore **0CBAh** in ciascun elemento del vettore; a tale proposito predisponiamo il seguente loop (per ciascuna istruzione viene indicato l'offset e il codice macchina):

```
006Eh  B8 0CBAh      mov      ax, 0CBAh               ; valore da caricare
0071h  BB 000Eh      mov      bx, offset vectorWord   ; bx punta al vettore
0074h  B9 0064h      mov      cx, 100                 ; contatore = 100
0077h                init_vector_loop:                ; etichetta di inizio loop
0077h  89 07h        mov      [bx], ax                ; carica l'elemento 
corrente
0079h  83 C3 02h     add      bx, 2                   ; prossimo elemento da 
caricare
007Ch  E2 F9h        loop     init_vector_loop        ; controllo del loop
007Eh  8B D8h        mov      bx, ax                  ; istruzione successiva
```

Le prime tre istruzioni eseguono una serie di inizializzazioni; il registro **AX** viene inizializzato con il valore **0CBAh** che verra' caricato in ogni elemento del vettore. Il registro **BX** viene fatto puntare a **vectorWord**; come si puo' notare **BX** viene inizializzato con il valore **000Eh** che e' proprio l'offset da cui parte il vettore. Siccome **vectorWord** e' formato da **100** elementi, il registro **CX** viene inizializzato con il valore **100**; questo significa che il loop verra' ripetuto **100** volte. L'etichetta **init_vector_loop** delimita l'inizio del loop e si trova all'offset **0077h** del blocco codice del nostro programma; come gia' sappiamo le etichette non hanno nessun codice macchina in quanto rappresentano solo dei punti di riferimento (offset) all'interno di un segmento di programma. La prima istruzione del loop carica il valore **0CBAh** nel primo elemento del vettore; osserviamo infatti che **BX** contiene l'offset **000Eh**, per cui l'istruzione:

`mov [bx], ax` trasferisce il contenuto a **16** bit di **AX** nella locazione di memoria **DS:000Eh**. La seconda istruzione del loop incrementa il registro puntatore **BX** di **2** in modo da farlo puntare al secondo elemento di **vectorWord**; dopo questo incremento **BX** punta infatti all'offset **0010h** del blocco dati. A questo punto la CPU incontra il codice macchina dell'istruzione **LOOP**; come si puo' notare, l'assembler ha generato il codice macchina **E2h** di **LOOP** seguito dal valore immediato **F9h**. Per i numeri con segno a **8** bit, **F9h** e' la rappresentazione in complemento a due del numero negativo **-7**; osserviamo infatti che l'etichetta **init_vector_loop** si trova all'offset **0077h**, precedendo di **7** byte l'offset **007Eh** dell'istruzione successiva a **LOOP**. Eseguendo l'istruzione **LOOP** la CPU decrementa **CX** di **1** ottenendo **CX=99**; siccome **CX** non si e' azzerato, il loop deve continuare. La CPU sottrae **7** byte all'offset **007Eh** e ottiene come risultato **0077h**; quest'offset viene caricato in **IP** e l'esecuzione salta a **CS:IP**. Dopo **99** iterazioni si ottiene **CX=1**; alla centesima iterazione **CX** si azzera e l'esecuzione passa all'offset **007Eh** ponendo fine quindi al loop.
Se vogliamo verificare gli effetti prodotti da questo loop, possiamo provare a visualizzare sullo schermo il contenuto degli elementi del vettore; utilizzando ad esempio la procedura **writeHex16** della libreria **EXELIB**, possiamo stampare i primi **10** elementi del vettore attraverso il seguente loop:

```
   mov      bx, offset vectorWord   ; bx punta all'inizio del vettore
   mov      dx, 0400h               ; riga/colonna iniziali per l'output
   mov      cx, 10                  ; stampa dei primi 10 elementi del vettore
print_vector_loop:                  ; etichetta di inizio loop
   mov      ax, [bx]                ; ax = elemento da stampare
   call     writeHex16              ; stampa dell'elemento
   add      bx, 2                   ; aggiornamento puntatore al vettore
   inc      dh                      ; incremento riga di output
   loop     print_vector_loop       ; controllo del loop
```

Questo esempio tiene conto del fatto che la procedura **writeHex16** preserva il contenuto di tutti i registri che utilizza; in caso contrario, il loop potrebbe anche mandare in crash il nostro programma. Questo accade in particolare quando all'interno di un loop si chiama una procedura che modifica il contenuto di **CX**; una procedura che ad esempio termina ogni volta con **CX=4**, impedisce al contatore di azzerarsi e porta quindi alla conseguenza di un loop infinito. Se la procedura **writeHex16** non avesse preservato il contenuto dei registri **BX**, **CX** e **DX**, avremmo dovuto scrivere:

```
push     bx          ; salva bx
push     cx          ; salva cx
push     dx          ; salva dx
call     writeHex16  ; chiama la procedura
pop      dx          ; ripristina dx
pop      cx          ; ripristina cx
pop      bx          ; ripristina bx
```

Ricordiamoci sempre di effettuare le estrazioni dallo stack in ordine inverso rispetto agli inserimenti; nel caso dell'esempio possiamo osservare infatti che dopo i tre inserimenti, il contenuto del registro **DX** si viene a trovare in cima allo stack, seguito dal contenuto di **CX** e poi dal contenuto di **BX**.

Tornando all'istruzione **LOOP**, dalle considerazioni esposte si deduce che l'etichetta che delimita l'inizio del loop deve trovarsi ad una distanza massima compresa tra **-128** e **+127** byte rispetto all'offset dell'istruzione immediatamente successiva a **LOOP**; in un caso del genere, il salto compiuto da **LOOP** viene chiamato **short jump** (salto corto). Lo short jump e' quindi piu' restrittivo del near jump e a maggior ragione non altera il contenuto di **CS**; in sostanza questo

significa che un loop deve svolgersi obbligatoriamente all'interno dello stesso segmento di programma. Si potrebbe pensare che lo short jump sia troppo limitato per la gestione di loop molto complessi; bisogna tener presente pero' che nella stragrande maggioranza dei casi si ha bisogno di loop molto piccoli e veloci. La presenza nei nostri programmi di loop molto grandi e contorti e' indice di un pessimo stile di programmazione; in ogni caso, nel seguito di questo capitolo vengono illustrate apposite istruzioni che consentono di superare i limiti imposti dallo short jump.

In alternativa a **LOOP**, le CPU **80x86** mettono a disposizione anche le due istruzioni **LOOPE/LOOPZ** e **LOOPNE/LOOPNZ**; queste istruzioni vengono utilizzate per implementare metodi piu' sofisticati per il controllo di un loop. Il codice macchina di **LOOPE/LOOPZ** e' formato dall'opcode **11100001b = E1h** seguito da un valore immediato **DISP** a **8** bit, mentre il codice macchina dell'istruzione **LOOPNE/LOOPNZ** e' formato dall'opcode **11100000b = E0h** seguito da un valore immediato **DISP** a **8** bit; il significato di **DISP** e' lo stesso gia' illustrato per l'istruzione **LOOP**.
Abbiamo visto che quando la CPU incontra il codice macchina dell'istruzione **LOOP** decrementa **CX** di **1** e ripete il loop solo se **CX** e' diverso da zero; l'effetto prodotto dall'esecuzione delle istruzioni **LOOPE/LOOPZ** e **LOOPNE/LOOPNZ** e' del tutto simile, e si distingue solo per il fatto che il controllo del loop viene effettuato attraverso **CX** e **ZF** (Zero Flag). Quando la CPU incontra il codice macchina dell'istruzione **LOOPE/LOOPZ**, decrementa **CX** di **1** e ripete il loop solo se **CX** e' diverso da zero e **ZF=1**; analogamente, quando la CPU incontra il codice macchina dell'istruzione **LOOPNE/LOOPNZ**, decrementa **CX** di **1** e ripete il loop solo se **CX** e' diverso da zero e **ZF=0**. Come si puo' notare, ciascuna di queste istruzioni dispone di due nomi equivalenti tra loro; lo scopo di questa "abbondanza" e' quello di dare la possibilita' al programmatore di scegliere la forma piu' espressiva dell'istruzione in relazione al contesto del programma che sta scrivendo.
Queste due istruzioni si rivelano molto utili quando abbiamo la necessita' di implementare un loop la cui terminazione e' legata non solo all'azzerarsi del contatore, ma anche al risultato di una operazione logico aritmetica che modifica **ZF**; questa operazione deve comparire come ultima istruzione del corpo del loop in modo che **LOOPE/LOOPZ** o **LOOPNE/LOOPNZ** possano controllare immediatamente gli effetti prodotti dall'operazione stessa su **ZF**. Vediamo un esempio molto semplice che illustra l'utilizzo di **LOOPNE/LOOPNZ**; supponiamo di aver definito la seguente stringa:

```
stringaC db 'stringadaconvertire', 0.
```

Si tratta di una stringa **C** terminata come al solito da uno zero; la stringa e' formata esclusivamente da lettere minuscole e non contiene ne spazi ne segni di punteggiatura. Supponiamo ora di voler convertire la stringa in maiuscolo; per fare questo, la prima cosa da osservare e' che i codici ASCII delle lettere maiuscole vanno da **65** a **90**, mentre i codici **ASCII** delle lettere minuscole vanno da **97** a **122**. La distanza costante che separa una lettera maiuscola dalla corrispondente lettera minuscola e' **32**; a questo punto appare abbastanza evidente che per convertire una lettera minuscola nella corrispondente lettera maiuscola basta sottrarre **32** al suo codice **ASCII**. Possiamo utilizzare allora un loop che scorre tutti gli elementi della stringa sottraendo **32** a ciascuno di essi; la condizione di uscita dal loop e' rappresentata dal raggiungimento dello zero finale della stringa **C**. Un primo metodo che si puo' utilizzare consiste nel determinare in anticipo la lunghezza della stringa caricando poi questa lunghezza nel contatore **CX**; in questo caso il loop puo' essere controllato dalla solita istruzione **LOOP**.

Un secondo metodo consiste nel verificare all'interno del loop la condizione di fine stringa, cioe' il raggiungimento dello zero finale, e questo puo' essere fatto con l'ausilio dell'istruzione **LOOPNE/LOOPNZ**; vediamo appunto un algoritmo che segue il secondo metodo:

```
   mov      bx, offset stringaC     ; bx punta all'inizio della stringa
   mov      cx, 0FFFFh              ; lunghezza massima per la stringa
convert_string_loop:                ; etichetta di inizio loop
   sub      byte ptr [bx], 32       ; conversione minuscolo-maiuscolo
   inc      bx                      ; aggiornamento puntatore
   cmp      byte ptr [bx], 0        ; fine stringa ?
   loopne   convert_string_loop     ; controllo del loop
```

Il contatore **CX** viene inizializzato con un valore molto alto in quanto l'uscita dal loop dipende principalmente dal raggiungimento della fine della stringa; la prima istruzione del corpo del loop sottrae **32** alla locazione di memoria puntata da **DS:BX**. Il registro **BX** punta all'inizio di **stringaC** per cui **[BX]** rappresenta il primo elemento della stringa; il type override **byte ptr** e' necessario per consentire all'assembler di determinare l'ampiezza in bit degli operandi della sottrazione. La seconda istruzione fa puntare **BX** all'elemento successivo della stringa; **BX** viene incrementato di **1** perche' la stringa come sappiamo e' un vettore di byte. La terza istruzione esegue una comparazione tra il contenuto della locazione di memoria **DS:BX** e **0**; come gia' sappiamo, l'istruzione **CMP** sottrae l'operando sorgente dall'operando destinazione modificando diversi flags (tra i quali **ZF**) e preservando l'operando destinazione. A questo punto l'istruzione **LOOPNE** controlla il loop in base al contenuto di **CX** e di **ZF**; il ruolo piu' importante viene svolto da **ZF** in quanto **CX** e' stato inizializzato con un valore molto alto e difficilmente si puo' azzerare prima del raggiungimento della fine della stringa. Se la fine della stringa non e' stata ancora raggiunta, l'istruzione **CMP** fornisce un risultato diverso da **0**; si ottiene quindi **ZF=0** e il loop viene ripetuto. Se invece e' stata raggiunta la fine della stringa, **CMP** calcola **0-0=0**; si ottiene quindi **ZF=1** e il loop termina preservando anche lo zero finale della stringa. A questo punto, se proviamo a stampare la stringa con la procedura **writeString** della libreria **EXELIB** otteniamo la visualizzazione sullo schermo di: `STRINGADACONVERTIRE`.
L'algoritmo appena illustrato e' molto semplice in quanto il suo scopo e' solamente quello di illustrare il funzionamento delle istruzioni **LOOPE/LOOPZ** e **LOOPNE/LOOPNZ**; questo algoritmo fallisce nel caso in cui la stringa contenga qualche lettera maiuscola, spazi, segni di punteggiatura, etc. Un'altro caso che provoca il fallimento di questo algoritmo e' rappresentato da una stringa di lunghezza nulla; piu' avanti vengono illustrate altre istruzioni che ci permettono di risolvere tutti questi problemi.

Tutte le considerazioni svolte si riferiscono come al solito alla modalita' reale **8086**; inoltre stiamo facendo riferimento a programmi costituiti da segmenti con attributo **SIZE** di tipo **USE16**. In questo caso, in relazione ai loop, il comportamento predefinito della CPU consiste nell'utilizzare **CX** come registro contatore; questo significa che un loop puo' essere ripetuto al massimo **65535** volte. Per poter utilizzare **ECX** come registro contatore, con la possibilita' quindi di ripetere un loop sino a **4294967295** volte, e' necessario abilitare il set di istruzioni a **32** bit e definire segmenti di programma con attributo **SIZE** di tipo **USE32**; questi aspetti vengono illustrati nella sezione **Modalita' Protetta**.
L'esecuzione delle istruzioni **LOOP**, **LOOPE/LOOPZ** e **LOOPNE/LOOPNZ**, non produce nessun effetto sul **Flags Register**.

L'istruzione JMP.

Con il mnemonico **JMP** si indica l'istruzione **unconditional JuMP** (salto incondizionato); questa istruzione viene utilizzata per costringere la CPU a saltare senza condizioni ad un'altro punto del programma. E' consentito sia il salto intrasegmento che il salto intersegmento; entrambi i salti possono essere diretti o indiretti e inoltre, il salto diretto intrasegmento puo' essere di tipo **NEAR** o **SHORT**. In sostanza, l'istruzione **JMP** e' molto simile all'istruzione **CALL**, con la differenza che **JMP** non salva nello stack nessun indirizzo di ritorno; il compito di **JMP** infatti e' semplicemente quello di effettuare un salto incondizionato ad un determinato indirizzo di memoria. Analizziamo i vari casi che si possono presentare.
1) Unconditional short jump direct within segment (salto corto incondizionato diretto intrasegmento); il codice macchina di questa istruzione e' formato dall'opcode **11101011b = EBh** seguito da un valore immediato **DISP** a **8** bit. Quando la CPU incontra questo codice macchina, somma algebricamente **DISP** all'offset dell'istruzione successiva a **JMP** ottenendo l'offset che rappresenta la destinazione dello short jump. Vediamo un esempio pratico:

```
0044h  EB 22h     jmp      salto_corto    ; salta a salto_corto
0046h  8B D8h     mov      bx, ax         ; istruzione successiva a jmp
.................................         ; altre istruzioni
0068h             salto_corto:            ; destinazione del salto
```

Incontrando il codice macchina **EB 22h** la CPU somma **22h** all'offset **0046h** ottenendo **0068h**; quest'offset viene caricato in **IP** e l'esecuzione salta a **CS:IP** senza alterare il contenuto di **CS**. Gli assembler ottimizzanti come **MASM** e **TASM** sono in grado di rilevare automaticamente il tipo di salto piu' opportuno in modo da generare il codice macchina piu' compatto possibile; in ogni caso, se vogliamo rendere esplicita la richiesta di uno short jump possiamo usare il type override **short** scrivendo:

```
jmp short salto_corto.
```

2) Unconditional near jump direct within segment (salto near incondizionato diretto intrasegmento); il codice macchina di questa istruzione e' formato dall'opcode **11101001b = E9h** seguito da un valore immediato **DISP** a **16** bit. Quando la CPU incontra questo codice macchina, somma algebricamente **DISP** all'offset dell'istruzione successiva a **JMP** ottenendo l'offset che rappresenta la destinazione del near jump. Vediamo un esempio pratico:

```
0044h  E9 0202h   jmp      salto_near     ; salta a salto_near
0047h  8B D8h     mov      bx, ax         ; istruzione successiva a jmp
.................................         ; altre istruzioni
0249h             salto_near:             ; destinazione del salto
```

Incontrando il codice macchina **E9 0202h** la CPU somma **0202h** all'offset **0047h** ottenendo **0249h**; quest'offset viene caricato in **IP** e l'esecuzione salta a **CS:IP** senza alterare il contenuto di **CS**. In questo caso non c'e' bisogno del type override in quanto il salto supera il limite di **+127** byte previsto per lo short jump; l'assembler utilizza quindi un operando **DISP** a **16** bit che consente di saltare in qualunque altro punto del segmento di programma contenente l'istruzione **JMP**.

3) Unconditional near jump indirect within segment (salto near incondizionato indiretto intrasegmento); il codice macchina di questa istruzione e' formato dagli opcodes **11111111b = FFh** e **mod_100_r/m**. Il secondo opcode codifica un registro a **16** bit o una locazione di memoria a **16** bit che contengono l'offset a cui **JMP** deve saltare. Quando la CPU incontra questo codice macchina, carica l'offset specificato in **IP** e salta a **CS:IP**. Vediamo un esempio pratico nel quale

l'offset a cui saltare viene caricato nella variabile a **16** bit **varTarget**; si assume che questa variabile sia stata definita all'offset **0004h** del segmento dati del programma.

```
0044h  C7 06 0004 0070h    mov      varTarget, offset salto_near  ; offset a cui
saltare
004Ah  FF 26 0004h         jmp      varTarget                     ; salta a
salto_near
004Eh  8B D8h              mov      bx, ax                        ; istruzione
successiva a jmp
...........................................                       ; altre
istruzioni
0070h                      salto_near:                            ; destinazione
del salto
```

La prima istruzione e' un trasferimento dati da **imm16** a **mem16**; questa istruzione trasferisce l'offset **0070h** dell'etichetta **salto_near** all'indirizzo **DS:0004h** dove e' stata definita **varTarget**. Per la seconda istruzione l'assembler ha generato il primo opcode **FFh** che insieme al campo **reg=100** indica un salto near indiretto intrasegmento; l'offset a cui saltare e' contenuto nella variabile **varTarget** per cui l'assembler pone **mod_r/m=00_110** ottenendo il secondo opcode **26h** seguito poi dall'offset **0004h** di **varTarget**. Incontrando il codice macchina **FF 26 0004h** la CPU accede a **DS:0004h**, legge il contenuto a **16** bit **0070h**, lo carica in **IP** e salta a **CS:IP**; anche in questo caso si nota che il contenuto di **CS** non viene alterato. L'offset di **salto_near** puo' essere caricato anche in un registro a **16** bit; usando ad esempio **BX** possiamo scrivere:
`jmp bx`.

4) Unconditional far jump direct intersegment (salto far incondizionato diretto intersegmento); il codice macchina di questa istruzione e' formato dall'opcode **11101010b = EAh** seguito da una coppia **SEG:OFFSET** da **16+16** bit che rappresenta l'indirizzo **FAR** a cui saltare. Quando la CPU incontra questo codice macchina, carica la coppia **SEG:OFFSET** in **CS:IP** e salta a **CS:IP**; vediamo un esempio pratico che illustra un salto far diretto da **CODESEGM** ad una etichetta **salto_far** definita all'offset **0006h** di **CODESEGM2**:

```
0044h  EA 00000006h  jmp   far ptr salto_far ; salta a salto_far
```

Come si puo' notare l'assembler ha generato l'opcode **EAh** seguito da un valore immediato a **32** bit **00000006h** che deve essere interpretato come coppia **0000h:0006h**; come al solito la componente **SEG** vale simbolicamente zero e verra' poi rilocata dal **SO** al momento di caricare il programma in memoria.
Incontrando il codice macchina **EA 00000006h** la CPU carica la coppia **CODESEGM2:0006h** in **CS:IP** e salta a **CS:IP**; trattandosi di un indirizzo **FAR**, viene modificato anche il contenuto di **CS**. Subito dopo il far jump, ci ritroviamo nel segmento **CODESEGM2**; chiaramente, se vogliamo tornare indietro, dobbiamo effettuare un'altro far jump verso **CODESEGM**.

5) Unconditional far jump indirect intersegment (salto far incondizionato indiretto intersegmento); il codice macchina di questa istruzione e' formato dagli opcodes **11111111b = FFh** e **mod_101_r/m**. Il secondo opcode codifica una locazione di memoria a **16+16** bit che contiene una coppia **SEG:OFFSET** a cui **JMP** deve saltare. Quando la CPU incontra questo codice macchina, carica la coppia **SEG:OFFSET** in **CS:IP** e salta a **CS:IP**. Vediamo un esempio pratico che illustra un salto far indiretto da **CODESEGM** ad una etichetta **salto_far** definita all'offset **0006h** di **CODESEGM2**; l'indirizzo **FAR** di **salto_far** viene caricato nella variabile a **32** bit **varTarget** definita all'offset **0004h** del segmento dati del programma.

```
0044h  C7 06 0004 0006h    mov      word ptr varTarget[0], offset salto_far
004Ah  C7 06 0006 0000h    mov      word ptr varTarget[2], seg      salto_far
0050h  FF 2E 0004h         jmp      varTarget
```

La prima istruzione trasferisce l'offset **0006h** dell'etichetta **salto_far** all'indirizzo **DS:0004h** dove si trova la word meno significativa di **varTarget**; la seconda istruzione trasferisce il segmento **0000h** (da rilocare) dell'etichetta **salto_far** all'indirizzo **DS:0006h** dove si trova la word piu' significativa di **varTarget** (convenzione **LITTLE-ENDIAN**). Per la terza istruzione l'assembler ha generato il primo opcode **FFh** che insieme al campo **reg=101** indica un salto far indiretto intersegmento; l'indirizzo a cui saltare e' contenuto nella variabile **varTarget** per cui l'assembler pone **mod_r/m=00_110** ottenendo il secondo opcode **2Eh** seguito poi dall'offset **0004h** di **varTarget**.
Incontrando il codice macchina **FF 2E 0004h** la CPU accede a **DS:0004h**, legge il contenuto a **32** bit **CODESEGM2:0006h**, lo carica in **CS:IP** e salta a **CS:IP**; trattandosi di un indirizzo **FAR**, viene modificato anche il contenuto di **CS**. Come per il far jump diretto, subito dopo il far jump indiretto, ci ritroviamo nel segmento **CODESEGM2**; se vogliamo tornare indietro, dobbiamo effettuare un'altro far jump verso **CODESEGM**.

L'istruzione **JMP** viene utilizzata dai linguaggi di alto livello per implementare la "famigerata" istruzione **GOTO**; questa istruzione viene letteralmente "criminalizzata" da quasi tutti i libri sulla programmazione (spesso copiati l'uno dall'altro). In effetti, in base ai concetti appena esposti, si capisce subito che e' meglio non abusare dell'istruzione **JMP**; in ogni caso, come viene mostrato piu' avanti, in certe situazioni l'istruzione **JMP** si rivela molto utile se non indispensabile.
In modalita' reale **8086** l'esecuzione dell'istruzione **JMP** non altera nessun campo del **Flags Register**.

Le istruzioni JCXZ e JECXZ.

Con il mnemonico **JCXZ** si indica l'istruzione **Jump short if CX is Zero** (salto corto se **CX=0**); il codice macchina di questa istruzione e' formato dall'opcode **11100011b = E3h** seguito da un valore immediato **DISP** a **8** bit che ha lo stesso significato gia' illustrato per l'istruzione **LOOP**. Quando la CPU incontra questo codice macchina esamina il contenuto di **CX**; se **CX** e' diverso da **0** l'esecuzione del programma prosegue con l'istruzione successiva a **JCXZ**. Se invece **CX=0** la CPU somma algebricamente **DISP** all'offset dell'istruzione successiva a **JCXZ** e ottiene un offset che rappresenta la destinazione dello short jump di **JCXZ**; quest'offset viene caricato in **IP** e l'esecuzione del programma salta a **CS:IP**.
L'istruzione **JCXZ** si rivela molto utile per evitare una situazione potenzialmente pericolosa che si viene a creare quando un loop parte con **CX=0**; questa situazione puo' verificarsi ad esempio quando **CX** viene inizializzato attraverso una operazione aritmetica che puo' produrre anche un risultato nullo. Analizziamo quello che succede in un caso del genere; la prima cosa da dire e' che come al solito, il loop viene eseguito almeno una volta. Quando la CPU arriva all'istruzione **LOOP** decrementa **CX** di **1** ottenendo **0 - 1 = FFFFh**; la CPU constatando che **CX** e' diverso da zero salta all'inizio del loop. In definitiva il loop viene ripetuto **65535 + 1 = 65536** volte; per evitare questa situazione possiamo scrivere:

```
   jcxz     end_loop                 ; salta il loop se CX = 0
start_loop:                          ; etichetta di inizio loop
   ................                  ; corpo del loop
   loop     start_loop               ; controllo del loop
end_loop:                            ; etichetta di fine loop
```

Come si puo' notare, se **CX=0** il loop viene aggirato con uno short jump compiuto da **JCXZ**; trattandosi appunto di short jump, l'etichetta **end_loop** deve trovarsi ad una distanza massima di **127** byte rispetto all'offset dell'etichetta **start_loop**.
In generale quindi il caso di **CX=0** all'inizio di un loop viene considerato come una situazione di errore; molti programmatori Assembly pero' utilizzano questo trucco quando hanno la necessita' di ripetere un loop proprio **65536** volte. Con i **16** bit del registro **CX** possiamo gestire il valore massimo **65535**; se vogliamo ripetere un loop **65536** volte possiamo usare quindi l'espediente appena descritto.
Abilitando il set di istruzioni a **32** bit possiamo usare anche l'istruzione **JECXZ** che lavora su **ECX**; il codice macchina di **JECXZ** e' lo stesso di **JCXZ**, preceduto pero' dall'**Address Size Prefix = 67h**.
L'esecuzione dell'istruzione **JCXZ/JEXCZ** non altera nessun campo del **Flags Register**; le CPU **80x86** non dispongono di analoghe istruzioni **JCXNZ/JEXCNZ**.

Le istruzioni Jcond.

Nei precedenti capitoli sono state illustrate numerose istruzioni la cui esecuzione modifica uno o piu' campi del **Flags Register**; questo aspetto assume una grande importanza per le istruzioni logico aritmetiche. Ogni volta che la CPU esegue un'istruzione di tipo logico aritmetico, modifica una serie di campi del **Flags Register**; in questo modo la CPU fornisce al programmatore informazioni estremamente dettagliate sulle caratteristiche del risultato ottenuto. Analizzando il contenuto dei flags modificati, il programmatore puo' valutare la situazione e prendere le decisioni piu' opportune; questa tecnica viene ampiamente sfruttata dai linguaggi di programmazione di alto livello per implementare le istruzioni decisionali come ad esempio **IF**, **THEN**, **ELSE**, etc.
Analizziamo in particolare il caso dell'istruzione **CMP** (compare two operands); questa istruzione consente di effettuare comparazioni numeriche tra l'operando sorgente (**SRC**) e l'operando destinazione (**DEST**). L'istruzione **CMP** sottrae **SRC** da **DEST**, modifica i flags **OF**, **SF**, **ZF**, **AF**, **PF**, **CF** e scarta il risultato preservando cosi' il contenuto di **DEST**; dall'analisi dei flags modificati il programmatore puo' risalire all'esito della comparazione. Nel caso della comparazione tra numeri senza segno abbiamo visto che e' necessario consultare **CF** e **ZF**; se **CF=1** la sottrazione ha richiesto un prestito e quindi **DEST** e' minore di **SRC**. Se **CF=0** bisogna consultare anche **ZF**; se **ZF=0** allora **DEST** e' maggiore di **SRC**, mentre se **ZF=1** la sottrazione ha prodotto un risultato nullo e quindi **DEST** e' uguale a **SRC**. Il concetto di controllo di flusso consiste proprio nel fatto che un programma puo' prendere delle decisioni che dipendono dal risultato prodotto da una determinata operazione; possiamo scrivere ad esempio un programma che a un certo punto puo' prendere tre strade diverse in base ai tre possibili risultati prodotti da una comparazione numerica.
In Assembly per poter effettuare queste scelte dobbiamo consultare ogni volta gli opportuni flags; non c'e' dubbio pero' che a lungo andare questa situazione puo' diventare abbastanza fastidiosa. Nel caso poi della comparazione tra numeri con segno, il lavoro che il programmatore deve svolgere comincia a diventare piuttosto pesante; in questo caso abbiamo visto che e' necessario consultare **CF**, **SF** e **ZF** (vedere il capitolo **Istruzioni aritmetiche**).
Fortunatamente le CPU della famiglia **80x86** dispongono di un'autentica marea di istruzioni che eseguono tutto questo lavoro per noi; si tratta delle istruzioni che effettuano i cosiddetti **conditional jumps** (salti condizionati). Tutte queste istruzioni vengono indicate simbolicamente con il mnemonico **Jcond** che significa **jump if condition is met** (salta se la condizione e' verificata); la verifica della condizione richiesta viene effettuata direttamente dalla CPU attraverso l'analisi del contenuto dei vari flags. Se la condizione richiesta non e' verificata, l'esecuzione prosegue con l'istruzione immediatamente successiva a **Jcond**; se invece la condizione richiesta e' verificata, l'esecuzione salta all'indirizzo specificato dall'operando dell'istruzione **Jcond**. Con il set di istruzioni a **16** bit l'operando di **Jcond** deve essere un valore immediato **DISP** a **8** bit; il significato di **DISP** e' lo stesso gia' illustrato in precedenza e consente quindi di effettuare solamente short

jumps. Se si dispone di una CPU **80386** o superiore, allora abilitando il set di istruzioni a **32** bit e' possibile effettuare anche near jumps; in questo caso, con i segmenti di programma aventi attributo **SIZE = USE16** si puo' utilizzare un valore immediato **DISP** a **8** o **16** bit. Con i segmenti di programma aventi attributo **SIZE = USE32** si puo' utilizzare un valore immediato **DISP** a **8**, **16** o **32** bit; le istruzioni **Jcond** non consentono invece far jumps.
L'uso corretto delle istruzioni **Jcond** consiste nello scrivere un'istruzione che modifica i flags, immediatamente seguita dall'opportuna istruzione **Jcond**; in questo modo la CPU puo' esaminare subito il **Flags Register** e comportarsi di conseguenza. Cominciamo ad analizzare le istruzioni **Jcond** partendo da quelle che fanno esplicito riferimento ai flags; molte istruzioni sono disponibili con due nomi diversi per consentire al programmatore di utilizzare la forma ritenuta piu' adatta al contesto del programma.

istruzione	condizione	effetto
JO	OF = 1 ?	salta se c'e' stato un overflow (OF = 1)
JC	CF = 1 ?	salta se c'e' stato un riporto/prestito (CF = 1)
JZ/JE	ZF = 1 ?	salta se il risultato e' zero (ZF = 1)
JS	SF = 1 ?	salta se il risultato e' negativo (SF = 1)
JP/JPE	PF = 1 ?	salta se la parita' e' pari (PF = 1)
JNO	OF = 0 ?	salta se non c'e' stato un overflow (OF = 0)
JNC	CF = 0 ?	salta se non c'e' stato un riporto/prestito (CF = 0)
JNZ/JNE	ZF = 0 ?	salta se il risultato non e' zero (ZF = 0)
JNS	SF = 0 ?	salta se il risultato e' positivo (SF = 0)
JNP/JPO	PF = 0 ?	salta se la parita' e' dispari (PF = 0)

L'utilizzo di queste istruzioni e' abbastanza intuitivo; supponiamo di avere la necessita' di saltare ad un'altro indirizzo nel caso in cui **AX=0**. Possiamo scrivere:

```
   test     ax, ax             ; ax and ax
   jz       nuovo_indirizzo    ; se ZF = 1 salta a nuovo_indirizzo
   mov      bx, ax             ; se ZF = 0 continua qui
```

Come gia' sappiamo, l'istruzione **TEST** esegue un **AND** logico tra **DEST** e **SRC**, modifica i flags e preserva il contenuto di **DEST**; utilizzando lo stesso registro per **DEST** e **SRC** otteniamo zero solo se tutti i bit del registro valgono zero.
Le istruzioni **Jcond** nonostante la loro semplicita' ci permettono di fare cose molto interessanti; l'unica limitazione e' rappresentata dalla fantasia del programmatore. Supponiamo ad esempio di voler implementare un loop che supera il raggio d'azione dello short jump e che utilizza **ECX** come contatore; abilitando il set di istruzioni a **32** bit possiamo scrivere:

```
   mov      ecx, 300000        ; 300000 iterazioni
start_loop:                    ; etichetta di inizio loop
   ....................        ; corpo del loop
   dec      ecx                ; decremento contatore
   jz       end_loop           ; se ECX = 0 esce dal loop
   jmp      start_loop         ; altrimenti ripete il loop
end_loop:
```

Al termine del corpo del loop decrementiamo il contatore e verifichiamo con **JZ** se **ECX** si e' azzerato; se **ECX=0** l'istruzione **JZ** esce dal loop compiendo uno short jump verso l'etichetta **end_loop**. Se invece **ECX** non si e' azzerato, ripetiamo il loop attraverso un near jump a **start_loop** effettuato dall'istruzione **JMP**; grazie a **JMP** possiamo saltare in qualsiasi altro punto del segmento di programma superando le limitazioni dell'istruzione **LOOP**. Questa tecnica viene largamente utilizzata dai linguaggi di alto livello per implementare i vari costrutti per il controllo del flusso (come ad esempio i cicli **FOR**); l'esempio appena visto dimostra inoltre che in determinate circostanze il salto incondizionato **JMP** si rivela non solo utile ma addirittura indispensabile.

Passiamo ora ad un'altro gruppo di istruzioni **Jcond** che non fanno esplicito riferimento ai flags; il riferimento e' pero' implicito in quanto si tratta di istruzioni che consentono di prendere decisioni sulla base del risultato prodotto da una operazione logico aritmetica. In sostanza, questo gruppo di istruzioni **Jcond** analizza lo stato assunto dai vari flags dopo un'operazione e ci mette a disposizione tutte le informazioni necessarie per gestire il risultato ottenuto; questo ci permette ad esempio di prendere decisioni in base al risultato di una comparazione evitando tutto il lavoro di consultazione dei vari flags. Come abbiamo visto nei precedenti capitoli, in questi casi e' molto importante distinguere tra numeri senza segno e numeri con segno; proprio per questo motivo, questo gruppo di istruzioni **Jcond** viene a sua volta suddiviso in due sottogruppi di istruzioni che operano appunto sui numeri senza segno e sui numeri con segno. Per gestire correttamente la situazione, il programmatore deve prendere confidenza con la terminologia imposta dalla **Intel**; purtroppo in alcuni casi questa terminologia puo' generare una certa confusione. Seguendo comunque la logica non si dovrebbero incontrare particolari difficolta'; un aspetto importantissimo di cui si deve tenere sempre presente, riguarda il fatto che in tutte le considerazioni che seguono il concetto di **maggiore** o **minore** e' sempre riferito all'operando destinazione.

Partiamo dalle istruzioni **Jcond** che operano sui numeri senza segno; in questo caso, i mnemonici delle varie istruzioni utilizzano le lettere **A** e **B** che significano rispettivamente: **above** (maggiore) e **below** (minore). Queste lettere vengono utilizzate in combinazione con **N** che sta per **not** (negazione) ed **E** che sta per **equal** (uguaglianza); possiamo dire quindi che **AE** significa **above or equal** (maggiore o uguale), cosi' come **NBE** significa **not below or equal** (ne minore ne uguale). Nella tabella che segue vengono utilizzati i simboli: < (minore), > (maggiore), <= (minore o uguale), >= (maggiore o uguale); vediamo quindi l'elenco completo di queste istruzioni:

Istruzioni Jcond per i numeri senza segno		
istruzione	**condizione**	**effetto**
JB/JNAE	(DEST < SRC) ?	salta se **DEST** e' minore di **SRC**
JNB/JAE	(DEST >= SRC) ?	salta se **DEST** e' maggiore o uguale a **SRC**
JBE/JNA	(DEST <= SRC) ?	salta se **DEST** e' minore o uguale a **SRC**
JNBE/JA	(DEST > SRC) ?	salta se **DEST** e' maggiore di **SRC**

Ciascuna istruzione ha un doppio nome in virtu' del fatto che una determinata relazione matematica puo' essere espressa in due modi differenti; nel caso ad esempio dell'istruzione **JB/JNAE**, dire **jump if below** (salta se **DEST** e' minore di **SRC**), equivale a dire **jump if not above or equal** (salta se **DEST** non e' ne maggiore ne uguale a **SRC**). Analogamente, nel caso dell'istruzione **JBE/JNA**, dire **jump if below or equal** (salta se **DEST** e' minore o uguale a **SRC**), equivale a dire **jump if not above** (salta se **DEST** non e' maggiore di **SRC**); naturalmente e' possibile utilizzare la forma che piu' si adatta ai gusti personali.

Passiamo ora alle istruzioni **Jcond** che operano sui numeri con segno; in questo caso, i mnemonici delle varie istruzioni utilizzano le lettere **G** e **L** che significano rispettivamente: **greater** (maggiore) e **less** (minore). Anche queste lettere vengono utilizzate in combinazione con **N** che sta per **not** (negazione) ed **E** che sta per **equal** (uguaglianza); possiamo dire quindi che **GE** significa **greater or equal** (maggiore o uguale), cosi' come **NLE** significa **not less or equal** (ne minore ne uguale).

Istruzioni Jcond per i numeri con segno		
istruzione	**condizione**	**effetto**
JL/JNGE	(DEST < SRC) ?	salta se **DEST** e' minore di **SRC**
JNL/JGE	(DEST >= SRC) ?	salta se **DEST** e' maggiore o uguale a **SRC**
JLE/JNG	(DEST <= SRC) ?	salta se **DEST** e' minore o uguale a **SRC**
JNLE/JG	(DEST > SRC) ?	salta se **DEST** e' maggiore di **SRC**

Anche in questo caso, ciascuna istruzione puo' essere utilizzata con due nomi equivalenti; nel caso ad esempio dell'istruzione **JL/JNGE**, dire **jump if less** (salta se **DEST** e' minore di **SRC**), equivale a dire **jump if not greater or equal** (salta se **DEST** non e' ne maggiore ne uguale a **SRC**).

Anche l'impiego di queste istruzioni appare abbastanza intuitivo; grazie alla conoscenza delle varie istruzioni **Jcond** e' possibile realizzare qualsiasi costrutto per il controllo del flusso di un programma. Un'aspetto importantissimo da considerare e' dato dal fatto che l'esecuzione di una qualsiasi istruzione **Jcond** non produce nessun effetto sul **Flags Register**; questo ci consente di disporre in successione due o piu' istruzioni **Jcond** in modo da effettuare scelte multiple.
Supponiamo di avere la seguente struttura **IF**, **ELSE IF**, **ELSE** in linguaggio **C**:

```
if (var1 > var2)             /* se var1 e' maggiore di var2 */
   printf("var1 > var2");    /* stampa questa stringa */
else if (var1 < var2)        /* se invece var1 e' minore di var2 */
   printf("var1 < var2");    /* stampa questa stringa */
else                         /* altrimenti (var1 = var2) */
   printf("var1 = var2");    /* stampa questa stringa */
```

Vogliamo convertire questa struttura in Assembly; prima di tutto definiamo tutte le variabili necessarie:

```
var1        dw    38000
var2        dw    24500
stringa1    db    'var1 > var2', 0
stringa2    db    'var1 < var2', 0
stringa3    db    'var1 = var2', 0
```

Supponendo di operare sui numeri senza segno possiamo scrivere:

```
   mov      dx, 0400h           ; riga/colonna di output della stringa
   mov      ax, var1            ; ax = var1
   cmp      ax, var2            ; confronta var1 con var2
   ja       short maggiore      ; se var1 > var2 salta a maggiore
   jb       short minore        ; se invece var1 < var2 salta a minore
uguale:                         ; altrimenti (var1 = var2) esegui uguale
   mov      bx, offset stringa3 ; bx punta a stringa3
   jmp      short stampa        ; esci dalla struttura
minore:
   mov      bx, offset stringa2 ; bx punta a stringa2
   jmp      short stampa        ; esci dalla struttura
maggiore:
   mov      bx, offset stringa1 ; bx punta a stringa1
stampa:                         ; fine struttura if-else if-else
   call     writeString         ; stampa la stringa
```

Come si puo' notare, ogni salto (condizionato o incondizionato) usa il type override **short** in modo da costringere l'assembler a generare il codice macchina piu' compatto possibile (due byte per ogni istruzione di salto); in ogni caso gli assembler ottimizzanti come **MASM** e **TASM** inserirebbero uno short jump anche senza i nostri consigli. L'etichetta **uguale** non e' necessaria ed e' stata inserita solo per motivi di chiarezza; ricordiamoci che le etichette sono solo dei punti di riferimento all'interno dei segmenti che le contengono, per cui ne possiamo inserire quante ne vogliamo in modo da rendere il codice piu' chiaro ed elegante. La tecnica che la CPU usa per eseguire istruzioni come **JA**, **JB**, etc e' proprio quella che e' stata descritta nel capitolo **Istruzioni aritmetiche** in riferimento all'istruzione **CMP**.

In modalita' reale a **16** bit, quando si realizzano strutture di controllo molto complesse, puo' presentarsi il problema di una o piu' istruzioni **Jcond** che si trovano a dover effettuare uno short jump verso un'etichetta troppo lontana; questo problema riguarda piu' in generale il caso in cui si abbia la necessita' di dover effettuare un salto intersegmento con una istruzione **Jcond**. Il problema si puo' risolvere in modo molto semplice; supponiamo di avere la seguente porzione di codice:

```
   test     ax, ax   ; ax = 0 ?
   jz       label1   ; salta a label1 se ax = 0
   mov      bx, ax   ; altrimenti continua qui
```

Se l'etichetta **label1** e' troppo lontana (set di istruzioni a **16** bit) oppure si trova in un'altro segmento di programma, adottando un ragionamento "inverso" possiamo scrivere:

```
   test     ax, ax   ; ax = 0 ?
   jnz      continua ; salta a continua se ax e' diverso da zero
   jmp      label1   ; salta a label1 se ax = 0
continua:
   mov      bx, ax
```

Come si puo' notare, la condizione **JZ** e' stata invertita in **JNZ**; se **JNZ** e' verificata viene compiuto uno short jump a **continua**. In caso contrario attraverso **JMP** si salta **label1** che puo' trovarsi nello stesso segmento di programma (short o near jump) o in un'altro segmento (far jump); questa tecnica prende il nome di **reverse logic** (logica inversa).

Come esempio finale riprendiamo il caso della conversione minuscolo-maiuscolo di una stringa **C**; vogliamo scrivere un algoritmo che puo' operare su stringhe di lunghezza nulla (formate solo dallo

zero finale), o su stringhe contenenti anche lettere maiuscole, segni di punteggiatura, spazi, etc.
Prima di tutto osserviamo che le lettere minuscole hanno codici **ASCII** che vanno da **97** a **122**; il nostro algoritmo deve quindi verificare se il carattere da convertire ricade in questo intervallo numerico. In caso affermativo la conversione avviene come al solito sottraendo **32** al codice **ASCII** del carattere stesso; in caso negativo si passa al prossimo carattere da esaminare. Per gestire anche le stringhe di lunghezza nulla la condizione di fine stringa deve essere testata all'inizio del loop; questo test serve in generale come condizione principale di uscita dal loop.
Partiamo dalla definizione della stringa da convertire:

```
stringaC db 'Stringa C da convertire!', 0.
```

A questo punto possiamo procedere con la scrittura dell'algoritmo di conversione; questa volta utilizziamo direttamente il nome **stringaC** per accedere alla stringa, e ci muoviamo da un elemento all'altro attraverso **BX**. Come gia' sappiamo, la notazione **stringaC[BX]** significa: accesso al contenuto dell'indirizzo che si ottiene sommando l'indirizzo iniziale di **stringaC** con il contenuto di **BX**; possiamo dire quindi che **stringaC[0]** accede al primo carattere della stringa, **stringaC[1]** accede al secondo carattere, etc.

```
   xor      bx, bx                     ; bx = 0 (primo carattere della stringa)
   mov      cx, 0FFFFh                 ; contatore = 65535
start_loop:                            ; inizio loop
   mov      al, stringaC[bx]           ; codice ASCII da esaminare
   test     al, al                     ; fine stringa ?
   jz       short exit_loop            ; se la stringa e' finita esci dal loop
   cmp      al, 97                     ; verifica limite inferiore
   jb       short next_char            ; non e' una lettera minuscola
   cmp      al, 122                    ; verifica limite superiore
   ja       short next_char            ; non e' una lettera minuscola
   sub      byte ptr stringaC[bx], 32  ; converte in maiuscolo
next_char:
   inc      bx                         ; prossimo carattere da esaminare
   loop     start_loop                 ; controllo del loop
exit_loop:                             ; uscita dal loop
```

Come regola generale, quando e' possibile si deve fare in modo che le varie istruzioni utilizzino operandi di tipo registro; in questo modo si ottiene la massima velocita' di esecuzione possibile.

Nota importante. In riferimento all'uso delle istruzioni **Jcond**, i manuali tecnici delle varie CPU consigliano vivamente di associare il salto alla condizione (**cond**) che ha la minore probabilita' di verificarsi; il perche' di questo consiglio e' abbastanza intuitivo. Quando la CPU sta eseguendo un'istruzione **Jcond**, ha gia' aggiornato **IP** in modo da farlo puntare all'istruzione successiva; se **cond** non e' verificata, l'esecuzione prosegue con l'istruzione successiva a **Jcond**, gia' puntata da **CS:IP**. Se invece **cond** e' verificata, la CPU e' costretta a modificare **IP** prima di saltare a **CS:IP** e questa operazione comporta chiaramente una certa perdita di tempo; possiamo dire quindi che associando il salto alla condizione piu' improbabile, nella maggioranza dei casi possiamo aspettarci un'esecuzione piu' veloce del nostro programma. Questo problema assume una certa gravita' nel caso di CPU dotate di **prefetch queue** (coda di precarica); la prefetch queue riduce enormemente i tempi di esecuzione delle varie istruzioni precaricate, purche' queste istruzioni debbano essere eseguite sequenzialmente. Se la CPU deve effettuare un salto, tutto il contenuto della prefetch queue non serve piu' a niente; in un caso del genere la CPU provvede a svuotare (**flush**) la prefetch queue e a ricaricarla con un nuovo blocco di istruzioni che iniziano dall'indirizzo di destinazione del salto. In questo caso la perdita di tempo e' veramente pesante; per alleviare questo grave problema le moderne CPU ricorrono a vari dispositivi hardware come la **cache memory**, la **doppia pipeline**, il **branch prediction**, etc. Questi concetti vengono illustrati nel capitolo **10** (**Struttura della CPU**).

Capitolo 21 - Istruzioni per il controllo della CPU

Come gia' sappiamo, quando si accende il computer, qualunque CPU della famiglia **80x86** viene inizializzata in modalita' di emulazione dell'**8086** (modalita' reale **8086**); questa fase di inizializzazione coinvolge anche il **Flags Register** con il flag **TF** che viene posto a **0**, il flag **IF** che viene posto a **1**, il flag **CF** che viene posto a **0**, etc. Questi ed altri aspetti nel loro insieme concorrono a determinare la modalita' operativa della CPU, cioe' il comportamento che la CPU segue per eseguire determinate istruzioni; a titolo di esempio, esistono particolari istruzioni i cui effetti dipendono dal contenuto di determinati campi del **Flags Register**. In certi casi il programmatore puo' avere la necessita' di alterare il modo di operare della CPU; per raggiungere questo obbiettivo, e' necessario ricorrere alle istruzioni per il controllo della CPU.

L'istruzione CLC.

Con il mnemonico **CLC** si indica l'istruzione **CLear Carry flag** (azzeramento del flag **CF**); subito dopo l'esecuzione di questa istruzione si ottiene **CF=0**. In gergo informatico il termine inglese **clear** (pulire) applicato ad un valore numerico, significa azzerare il valore stesso; "pulire" il Carry Flag significa quindi porre **CF=0**, cosi' come "pulire" il registro **AX** significa porre **AX=0**. Quando il valore numerico e' formato da un solo bit, si utilizzano anche i termini **set/reset**; il termine **set** (abilitare) applicato ad un bit significa porre il bit a **1**, mentre il termine **reset** (disabilitare) applicato ad un bit significa porre il bit a **0**.
L'istruzione **CLC** si ripercuote su tutte quelle istruzioni che vengono eseguite tenendo conto del flag **CF**; in particolare si possono citare le istruzioni **ADC** e **SBB**.
Il codice macchina dell'istruzione **CLC** e' formato dal solo opcode **11111000b = F8h**; l'esecuzione dell'istruzione **CLC** altera ovviamente il solo flag **CF** lasciando inalterati tutti gli altri flags.

L'istruzione STC.

Con il mnemonico **STC** si indica l'istruzione **SeT Carry flag** (abilitazione del flag **CF**); subito dopo l'esecuzione di questa istruzione si ottiene **CF=1**.
Il codice macchina dell'istruzione **STC** e' formato dal solo opcode **11111001b = F9h**; l'esecuzione dell'istruzione **STC** altera ovviamente il solo flag **CF** lasciando inalterati tutti gli altri flags.

L'istruzione CMC.

Con il mnemonico **CMC** si indica l'istruzione **CoMplement Carry flag** (complemento a uno del flag **CF**); subito dopo l'esecuzione di questa istruzione il contenuto del flag **CF** viene invertito. Se **CF** valeva inizialmente **0**, dopo l'esecuzione di **CMC** si ottiene **CF=1**; viceversa, se **CF** valeva inizialmente **1**, dopo l'esecuzione di **CMC** si ottiene **CF=0**.
Il codice macchina dell'istruzione **CMC** e' formato dal solo opcode **11110101b = F5h**; l'esecuzione dell'istruzione **CMC** altera ovviamente il solo flag **CF** lasciando inalterati tutti gli altri flags.

L'istruzione CLD.

Con il mnemonico **CLD** si indica l'istruzione **CLear Direction flag** (azzeramento del flag **DF**); subito dopo l'esecuzione di questa istruzione si ottiene **DF=0**. Il **Direction Flag** (flag di direzione) assume una notevole importanza per tutte le istruzioni di manipolazione delle stringhe che vengono illustrate nel capitolo successivo; per eseguire queste istruzioni la CPU utilizza implicitamente i registri puntatori **DI/EDI**, **SI/ESI** attraverso i quali vengono indirizzate le stringhe da manipolare.
L'esecuzione delle istruzioni per la manipolazione delle stringhe determina automaticamente l'aggiornamento dei registri puntatori **DI/EDI** e **SI/ESI**; se **DF=0** i puntatori vengono incrementati,

mentre se **DF=1** i puntatori vengono decrementati.
Il codice macchina dell'istruzione **CLD** e' formato dal solo opcode **11111100b = FCh**; l'esecuzione dell'istruzione **CLD** altera ovviamente il solo flag **DF** lasciando inalterati tutti gli altri flags.

L'istruzione STD.

Con il mnemonico **STD** si indica l'istruzione **SeT Direction flag** (abilitazione del flag **DF**); subito dopo l'esecuzione di questa istruzione si ottiene **DF=1**. In questo caso, eseguendo un'istruzione per la manipolazione delle stringhe, si ottiene il decremento del puntatore alla stringa stessa; questo significa che il puntatore viene posizionato in corrispondenza dell'elemento precedente della stringa.
Il codice macchina dell'istruzione **STD** e' formato dal solo opcode **11111101b = FDh**; l'esecuzione dell'istruzione **STD** altera ovviamente il solo flag **DF** lasciando inalterati tutti gli altri flags.

L'istruzione CLI.

Con il mnemonico **CLI** si indica l'istruzione **CLear Interrupt enable flag** (azzeramento del flag **IF**); subito dopo l'esecuzione di questa istruzione si ottiene **IF=0**. Azzerando questo flag si blocca l'elaborazione da parte della CPU delle cosiddette **maskable interrupts** (interruzioni mascherabili); tutte le interruzioni mascherabili vengono poste in stato di attesa finche' non si riporta **IF** a **1**. Come abbiamo visto nel precedente capitolo, per evitare la sovrapposizione delle interruzioni mascherabili la CPU pone **IF=0** ogni volta che deve chiamare una **ISR** (esecuzione di **INT**); al termine della **ISR** l'istruzione **IRET** ripristina il **Flags Register** riportando **IF** a **1**. Anche il programmatore puo' avere la necessita' di seguire un procedimento del genere; questo accade quando il programma che stiamo scrivendo deve eseguire un compito molto delicato come ad esempio il cronometraggio di un tempo molto piccolo (frazioni di secondo). In un caso del genere, il sopraggiungere di una interruzione mascherabile puo' alterare completamente il risultato prodotto dal cronometraggio stesso; per evitare questo inconveniente e' possibile disabilitare temporaneamente **IF** provvedendo poi alla sua riabilitazione. Come e' stato ampiamente detto nei precedenti capitoli, per garantire il corretto funzionamento del computer e' fondamentale riabilitare **IF** nel piu' breve tempo possibile.
Ponendo **IF=0** si bloccano solo le interruzioni mascherabili generate direttamente dall'hardware del computer (**hardware interrupts**); lo stato di **IF** non ha nessun effetto sulle interruzioni non mascherabili (**NMI**) che vengono generate nel caso di gravissimi problemi hardware verificatisi nel computer. Lo stato di **IF** non influisce nemmeno sulle interruzioni generate direttamente dai programmi (**software interrupts**); come sappiamo, queste interruzioni vengono generate in seguito all'esecuzione dell'istruzione **INT**.
Il codice macchina dell'istruzione **CLI** e' formato dal solo opcode **11111010b = FAh**; l'esecuzione dell'istruzione **CLI** altera ovviamente il solo flag **IF** lasciando inalterati tutti gli altri flags.

L'istruzione STI.

Con il mnemonico **STI** si indica l'istruzione **SeT Interrupt enable flag** (abilitazione del flag **IF**); subito dopo l'esecuzione di questa istruzione si ottiene **IF=1**. In questo modo si riattiva l'elaborazione da parte della CPU delle interruzioni mascherabili; quando si accende il computer, in fase di inizializzazione della CPU il flag **IF** viene ovviamente posto a **1**.
Il codice macchina dell'istruzione **STI** e' formato dal solo opcode **11111011b = FBh**; l'esecuzione dell'istruzione **STI** altera ovviamente il solo flag **IF** lasciando inalterati tutti gli altri flags.

L'istruzione WAIT.

Con il mnemonico **WAIT** si indica l'istruzione **WAIT until the BUSY# pin is high** (attendere finche' il coprocessore matematico e' occupato); l'esecuzione di questa istruzione sospende l'attivita' della CPU in attesa che il coprocessore matematico (**FPU**) porti a termine i calcoli che sta svolgendo. Le CPU, come tutti i circuiti integrati, comunicano con il mondo esterno attraverso una serie di terminali elettrici chiamati "piedini" o **pin**; uno di questi piedini viene chiamato **BUSY# pin** e connette elettricamente la CPU con la FPU. Attraverso il **BUSY# pin** la CPU e' in grado di conoscere istante per istante lo stato attivo/inattivo della FPU; quando la FPU e' attiva (cioe' sta eseguendo dei calcoli), il livello logico del **BUSY# pin** vale **1**. Viceversa, quando la FPU e' inattiva, il livello logico del **BUSY# pin** vale **0**; quando esegue l'istruzione **WAIT** la CPU verifica il livello logico presente nel **BUSY# pin**. Se il livello logico e' **0** la CPU prosegue normalmente il suo lavoro; se invece il livello logico e' **1** la CPU sospende la propria attivita' in attesa che lo stesso livello logico venga riportato a **0** dalla FPU.
Le istruzioni della FPU utilizzano dei mnemonici che iniziano tutti per **F** (**Fast** = veloce); ogni volta che la CPU incontra una di queste istruzioni, la "smista" alla FPU che provvede ad effettuare le necessarie elaborazioni. La FPU e' in grado di eseguire il proprio lavoro in modo indipendente rispetto alla CPU; in certi casi pero' il programmatore puo' aver bisogno di sincronizzare la FPU con la CPU. Per ottenere questa sincronizzazione si utilizza appunto l'istruzione **WAIT**; tutti questi concetti vengono esposti in dettaglio in un apposito capitolo della sezione **Assembly Avanzato**.
Il codice macchina dell'istruzione **WAIT** e' formato dal solo opcode **10011011b = 9Bh**; l'esecuzione dell'istruzione **WAIT** non produce nessun effetto sul **Flags Register**.

Capitolo 22 - Istruzioni per la manipolazione delle stringhe

In questo capitolo vengono esaminate una serie di potentissime istruzioni che la **Intel** definisce **istruzioni per la manipolazione delle stringhe** (o piu' brevemente **istruzioni per le stringhe**); in relazione a queste istruzioni il termine **stringa** deve essere inteso nel senso piu' generale possibile. In sostanza possiamo definire la stringa come un vettore di generici elementi, cioe' una sequenza di elementi tutti della stessa natura, disposti in memoria in modo consecutivo e contiguo; possiamo avere ad esempio stringhe formate da una sequenza di byte, da una sequenza di word, da una sequenza di double word, da una sequenza di quad word, etc. Il fatto che si tratti di un vettore impone che, come e' stato appena detto, i vari elementi si trovino disposti in memoria in modo consecutivo e contiguo; consideriamo ad esempio la seguente definizione presente all'offset **0000h** del blocco dati **DATASEGM** di un programma:

```
DATASEGM     SEGMENT PARA PUBLIC USE16 'DATA'

strDword     dd    03DF01ABh, 00BBA100h, 0388FF11h, 1A2F2244h
             dd    0011DD24h, 48AC2DFFh, 04BBCCDDh, 1234ABCDh

DATASEGM     ENDS
```

Quando l'assembler incontra questa definizione, predispone **8** locazioni di memoria consecutive e contigue a partire dall'offset **0000h** del blocco **DATASEGM**; ciascuna locazione di memoria occupa **4** byte e viene inizializzata con i valori specificati nella definizione della stringa. La Figura 1 mostra la disposizione che assumera' la stringa una volta che il programma verra' caricato in memoria.

Figura 1 - Stringa da 8 double word								
contenuto	1234ABCDh	04BBCCDDh	48AC2DFFh	0011DD24h	1A2F2244h	0388FF11h	00BBA100h	03DF01ABh
offset	001Ch	0018h	0014h	0010h	000Ch	0008h	0004h	0000h
indice	7	6	5	4	3	2	1	0

A ciascun elemento e' associato un **indice** (o **posizione**), un **indirizzo** (o **offset**) e un **contenuto**; l'**indice** di un elemento rappresenta la posizione dell'elemento stesso all'interno del vettore. Come al solito, nel rispetto della convenzione Assembly gli indici partono da zero; questo significa che l'elemento di posto **0** e' il primo elemento del vettore, l'elemento di posto **1** e' il secondo elemento del vettore, l'elemento di posto **2** e' il terzo elemento del vettore, etc. L'**indirizzo** di un elemento e' la sua distanza in byte (**offset**) dall'inizio del segmento di programma nel quale il vettore e' stato definito; in Figura 1 vediamo che l'elemento **0** si trova all'offset **0000h**, l'elemento **1** si trova all'offset **0004h**, l'elemento **2** si trova all'offset **0008h**, etc. Siccome ogni elemento occupa **4** byte, possiamo osservare che i vari elementi si susseguono a distanza di **4** byte l'uno dall'altro; in sostanza questo significa che partendo dall'offset del primo elemento, avanzando di **4** byte incontriamo il secondo elemento, avanzando di altri **4** byte incontriamo il terzo elemento e cosi' via. La memoria complessiva occupata da un vettore puo' essere ottenuta moltiplicando il numero degli elementi del vettore per la dimensione in byte di ogni elemento; nel caso di Figura 1 abbiamo un vettore che occupa una porzione di memoria da:
8 * 4 = 32 byte. Il **contenuto** di un elemento e' il valore binario che viene memorizzato nella locazione di memoria riservata all'elemento stesso; la Figura 1 mostra il contenuto dei vari elementi in notazione esadecimale. In virtu' del fatto che il computer rappresenta i numeri secondo il sistema posizionale arabo, e che questi numeri vengono disposti in memoria secondo la convenzione **LITTLE-ENDIAN**, e' importante schematizzare graficamente la memoria come una sequenza di locazioni i cui indirizzi crescono procedendo da destra verso sinistra; in questo modo il valore

numerico contenuto in ciascuna locazione ci appare disposto in modo naturale e quindi perfettamente leggibile. Se in Figura 1 avessimo disposto la sequenza delle locazioni di memoria con gli indirizzi crescenti da sinistra verso destra, avremmo ottenuto una rappresentazione poco comprensibile; ad esempio, il contenuto **03DF01ABh** dell'elemento **0** del vettore ci sarebbe apparso scritto come **AB01DF03h**.

Il nome **strDword** assegnato a questa stringa rappresenta l'offset iniziale (**0000h**) della stringa stessa, calcolato rispetto a **DATASEGM**; per accedere ai vari elementi della stringa possiamo utilizzare svariati metodi basati sui puntatori. Caricando ad esempio in **BX** l'offset **0000h** di **strDword**, possiamo accedere al primo elemento del vettore attraverso la notazione **[BX]**; per accedere agli elementi successivi ci basta incrementare il puntatore **BX** di **4** byte alla volta. Un'altro metodo consiste nell'usare ad esempio **BX** come registro base e **DI** come registro indice; caricando in **BX** l'offset iniziale **0000h** di **strDword** e in **DI** lo spiazzamento iniziale **0**, possiamo accedere al primo elemento del vettore attraverso la notazione **[BX+DI]**. Per accedere agli elementi successivi ci basta incrementare **DI** di **4** byte alla volta; al posto di **DI** si puo' utilizzare direttamente un valore immediato **0**, **4**, **8**, **12**, etc. Utilizzando inoltre il nome **strDword**, possiamo accedere ai vari elementi del vettore con la notazione:
strDword[DISP], dove **DISP** puo' essere un valore immediato o un registro puntatore; questa notazione significa: accedi all'indirizzo di memoria che si ottiene sommando **DISP** all'offset iniziale **0000h** di **strDword**. Il contenuto dei vari elementi e' rappresentato quindi dalle notazioni:
strDword[0], **strDword[4]**, **strDword[8]**, etc; come al solito, in Assembly il compito di calcolare i vari spiazzamenti spetta al programmatore. I compilatori dei linguaggi di alto livello invece svolgono questo lavoro per noi; in linguaggio **C** ad esempio, possiamo accedere ai vari elementi di **strDword** con le notazioni:
strDword[0], **strDword[1]**, **strDword[2]**, etc. Quando il compilatore **C** incontra ad esempio la notazione **strDword[3]** (quarto elemento di **strDword**), moltiplica l'indice **3** per la dimensione in byte (**4**) di ogni elemento del vettore ottenendo:
3 * 4 = 12; il compilatore accede quindi all'indirizzo di memoria che si ottiene sommando **12** byte all'offset iniziale **0000h** di **strDword**.
Per verificare in pratica tutte le considerazioni esposte scriviamo un apposito programma che visualizza sullo schermo l'indice, l'offset e il contenuto di ogni elemento della stringa **strDword**; utilizzando apposite procedure della libreria **EXELIB**, otteniamo il seguente loop:

```
   mov      bx, offset strDword        ; bx = offset iniziale di strDword
   mov      dh, 04h                    ; riga iniziale di output
   mov      cx, 8                      ; contatore = 8
print_str_loop:                        ; inizio loop
   mov      al, dh                     ; al = riga di output
   sub      al, 4                      ; indice = riga - 4
   mov      dl, 0                      ; colonna output indice
   call     writeUdec8                 ; visualizza l'indice
   mov      ax, bx                     ; ax = offset elemento
   add      dl, 8                      ; colonna output offset
   call     writeHex16                 ; visualizza l'offset
   mov      eax, [bx]                  ; eax = contenuto elemento
   add      dl, 8                      ; colonna output contenuto
   call     writeHex32                 ; visualizza il contenuto
   add      bx, 4                      ; aggiornamento puntatore
   inc      dh                         ; aggiornamento riga di output
   loop     print_str_loop             ; controllo del loop
```

E' importante ribadire ancora una volta che l'algoritmo appena illustrato tiene conto del fatto che le varie procedure chiamate all'interno del loop preservano il contenuto dei vari registri che utilizzano; in caso contrario il programmatore e' tenuto a prendere tutti i necessari provvedimenti.

Il funzionamento di questo algoritmo dovrebbe essere abbastanza intuitivo; l'indice dell'elemento viene calcolato sfruttando il numero della riga di output contenuto in **DH**. Il registro **DH** viene inizializzato con il valore **4** e viene incrementato di **1** ad ogni loop; di conseguenza, sottraendo **4** al contenuto di **DH** otteniamo la successione **0**, **1**, **2**, etc.
Un'altro aspetto sul quale il programmatore Assembly non deve avere il minimo dubbio riguarda i trasferimenti di dati che coinvolgono i puntatori; quando l'assembler incontra l'istruzione:
`mov al, [bx]`, capisce dalle dimensioni di **AL** che si tratta di un trasferimento dati a **8** bit e genera quindi il codice macchina che permette alla CPU di copiare in **AL** il contenuto a **8** bit della locazione **DS:BX**.
Quando l'assembler incontra l'istruzione:
`mov ax, [bx]`, capisce dalle dimensioni di **AX** che si tratta di un trasferimento dati a **16** bit e genera quindi il codice macchina che permette alla CPU di copiare in **AX** il contenuto a **16** bit della locazione **DS:BX**.
Quando l'assembler incontra l'istruzione:
`mov eax, [bx]`, capisce dalle dimensioni di **EAX** che si tratta di un trasferimento dati a **32** bit e genera quindi il codice macchina che permette alla CPU di copiare in **EAX** il contenuto a **32** bit della locazione **DS:BX**.
In sostanza, non bisogna fare confusione tra la dimensione in bit di un offset e la dimensione in bit del dato puntato da quell'offset; in modalita' reale **8086** un offset e' un numero intero senza segno a **16** bit, mentre un dato puntato da quell'offset puo' avere la dimensione di un byte, una word, una dword, etc. L'offset di un dato rappresenta l'indirizzo di memoria da cui inizia il dato stesso; la Figura 2 illustra ad esempio la disposizione in memoria (byte per byte) dei primi due elementi di **strDword**.

Figura 2 - strDword[0] e strDword[1]

contenuto	00h	BBh	A1h	00h	03h	DFh	01h	ABh
offset	0007h	0006h	0005h	0004h	0003h	0002h	0001h	0000h
indice	1				0			

Possiamo dire quindi che la prima dword (**03DF01ABh**) parte dall'offset **0000h** (offset iniziale) ed occupa in memoria tutti gli offset che vanno da **0000h** a **0003h**; analogamente, la seconda dword (**00BBA100h**) parte dall'offset **0004h** (offset iniziale) ed occupa in memoria tutti gli offset che vanno da **0004h** a **0007h**. Supponendo che **BX** contenga ad esempio l'offset **0004h**, il trasferimento dati a **32** bit da **[BX]** a **EAX** consiste nel copiare il byte in posizione **DS:0004h** (**00h**) nel primo byte di **EAX**, il byte in posizione **DS:0005h** (**A1h**) nel secondo byte di **EAX**, il byte in posizione **DS:0006h** (**BBh**) nel terzo byte di **EAX** e il byte in posizione **DS:0007h** (**00h**) nel quarto byte di **EAX**; al termine del trasferimento dati **EAX** contiene il valore **00BBA100h**.

Le istruzioni illustrate in questo capitolo operano sulle stringhe di byte, sulle stringhe di word e sulle stringhe di dword (solo CPU **80386** e superiori); prima di iniziare a conoscere queste istruzioni, analizziamo un caso molto particolare per le stringhe che assume una importanza enorme nel campo della programmazione.

Stringhe alfanumeriche generiche, stringhe C e stringhe Pascal.

Abbiamo visto quindi che le stringhe generiche possono essere definite come sequenze di elementi, tutti della stessa natura, disposti in memoria in modo consecutivo e contiguo; dal punto di vista del computer, il contenuto di ciascun elemento della stringa non e' altro che un normalissimo numero binario rappresentato da una sequenza di livelli logici. E' compito del programmatore associare a questo numero binario un significato ben preciso; nel caso ad esempio del vettore **strDword** di Figura 1, i valori a **32** bit contenuti in ciascun elemento potrebbero rappresentare numeri reali in formato **IEEE** a **32** bit (**short real**).
Consideriamo ora il caso particolare delle stringhe di byte; ciascun elemento appartenente ad una stringa di byte contiene quindi un generico valore binario a **8** bit compreso tra **00000000b** e **11111111b**. Queste normalissime stringhe diventano speciali nel momento in cui associamo il contenuto di ogni loro elemento ad un codice ASCII a **8** bit; in questo caso si parla di **stringhe alfanumeriche** o **stringhe ASCII** per indicare il fatto che queste stringhe contengono una sequenza di lettere dell'alfabeto, cifre decimali, spazi, segni di punteggiatura e altri simboli appartenenti al set di codici **ASCII**. Le stringhe **ASCII** sono talmente importanti che tutti i linguaggi di programmazione, compreso l'Assembly, le trattano come un vero e proprio tipo di dato predefinito; consideriamo ad esempio la seguente definizione di una generica stringa di byte in Assembly:

```
strByte  db    53h, 74h, 72h, 69h, 6Eh, 67h, 61h, 20h, 41h, 53h, 43h, 49h, 49h
```

Incontrando questa definizione, l'Assembly riserva a **strByte** **13** locazioni di memoria da **1** byte ciascuna per un totale di **13** byte; ogni locazione di memoria viene inizializzata con il valore indicato nella definizione di **strByte**. Supponendo che **strByte** sia stata definita a partire dall'offset **0018h** del blocco **DATASEGM**, quando il programma viene caricato in memoria la stringa assume la disposizione mostrata in Figura 3:

Figura 3 - Stringa da 10 byte													
contenuto	49h	49h	43h	53h	41h	20h	61h	67h	6Eh	69h	72h	74h	53h
offset	0024h	0023h	0022h	0021h	0020h	001Fh	001Eh	001Dh	001Ch	001Bh	001Ah	0019h	0018h
indice	12	11	10	9	8	7	6	5	4	3	2	1	0

Come al solito, dal punto di vista del computer il contenuto di ciascun elemento di questa stringa non e' altro che un generico numero binario a **8**; dal nostro punto di vista invece il contenuto di ciascun elemento di **strByte** puo' rappresentare ad esempio un numero senza segno a **8** bit, un numero con segno a **8** bit, un colore di un pixel dello schermo, un codice **ASCII**, etc: Analizziamo proprio il caso in cui **strByte** rappresenti una stringa di codici **ASCII** a **8** bit; associando a ciascun codice **ASCII** il corrispondente simbolo e scrivendo in sequenza i vari simboli otteniamo:
`Stringa ASCII.`
Tutti i linguaggi di programmazione, compreso l'Assembly, ci permettono di definire questo tipo di stringhe in modo molto semplice e compatto; in Assembly volendo trattare **strByte** come stringa alfanumerica, possiamo scrivere la seguente definizione:
`strByte db 'Stringa ASCII'.`
Quando l'assembler incontra questa definizione, si comporta esattamente come viene illustrato in Figura 3; in pratica l'assembler riserva **13** locazioni di memoria consecutive e contigue ciascuna delle quali occupa **1** byte. In queste **13** locazioni di memoria l'assembler carica i **13** codici **ASCII** associati ai **13** simboli presenti nella definizione di **strByte**; la stringa **strByte** viene quindi disposta in memoria come una normalissima stringa di byte. Per verificare questi concetti proviamo a stampare sullo schermo tutte le informazioni relative a ciascun elemento di **strByte**, e cioe' l'indice, l'offset, il contenuto in forma numerica e il contenuto in forma simbolica di ciascun elemento della

stringa; per stampare i vari simboli **ASCII** utilizziamo la procedura **writeString** che richiede in **BX** l'offset di una stringa terminata da uno **0**. A tale proposito definiamo un'apposita variabile **strChar** formata da due byte; il primo byte contiene il codice **ASCII** del simbolo da stampare mentre il secondo byte vale zero. Un'altro problema che dobbiamo affrontare riguarda il calcolo della lunghezza della stringa; questa lunghezza ci serve per inizializzare il contatore **CX** del loop. Un metodo molto usato dai programmatori Assembly consiste nell'utilizzare il **location counter** (**$**) che ci permette di ottenere la lunghezza di una stringa in fase di assemblaggio del programma; in definitiva il blocco dati del nostro programma assume la seguente struttura:

```
DATASEGM    SEGMENT PARA PUBLIC USE16 'DATA'

strChar     db    0, 0
strByte     db    'Stringa ASCII'
STRBYTELEN  =     $ - offset StrByte

DATASEGM    ENDS
```

Come gia' sappiamo, in fase di assemblaggio di un programma Assembly, il simbolo **$** (dollaro) rappresenta l'offset corrente all'interno del segmento di programma che l'assembler sta esaminando, cioe' la distanza di **$** dall'inizio del segmento di programma; osserviamo ora che **strChar** si trova all'offset **0000h** di **DATASEGM**, **strByte** si trova all'offset **0002h**, mentre **$** si trova all'offset **000Fh**, per cui il simbolo **$** in quella posizione rappresenta la componente **OFFSET = 000Fh** dell'indirizzo **DATASEGM:000Fh**. Quando l'assembler incontra la dichiarazione della costante: `STRBYTELEN = $ - offset strByte`, sottrae l'offset **0002h** di **strByte** dall'offset **000Fh** di **$** e ottiene quindi **000Dh = 13d**; questo valore viene assegnato alla costante **STRBYTELEN** e coincide proprio con la lunghezza in byte della stringa.
A questo punto abbiamo a disposizione tutto cio' che ci serve per poter visualizzare sullo schermo tutti i dettagli relativi a **strByte**; possiamo procedere quindi con la scrittura del seguente loop:

```
   mov      bx, offset strChar          ; bx punta a strChar
   mov      di, offset strByte          ; di punta a strByte
   mov      dh, 04h                     ; riga iniziale di output
   mov      cx, STRBYTELEN              ; contatore = STRBYTELEN
print_str_loop:                         ; inizio loop
   mov      al, dh                      ; al = riga di output
   sub      al, 4                       ; indice = riga - 4
   mov      dl, 0                       ; colonna output indice
   call     writeUdec8                  ; visualizza l'indice
   mov      ax, di                      ; ax = offset elemento
   add      dl, 8                       ; colonna output offset
   call     writeHex16                  ; visualizza l'offset
   mov      al, [di]                    ; al = codice ASCII
   add      dl, 8                       ; colonna output codice ASCII
   call     writeHex8                   ; visualizza il codice ASCII
   mov      [bx], al                    ; strChar = al, 0
   add      dl, 8                       ; colonna di output simbolo ASCII
   call     writeString                 ; visualizza il simbolo ASCII
   inc      di                          ; aggiornamento puntatore a strByte
   inc      dh                          ; aggiornamento riga di output
   loop     print_str_loop              ; controllo del loop
```

Questa volta utilizziamo i due registri puntatori **BX** e **DI**; il registro **BX** punta all'inizio di **strChar**, mentre **DI** punta all'inizio di **strByte**. All'interno del loop stampiamo i vari dettagli relativi ad ogni elemento di **strByte**; l'indice dell'elemento viene ricavato come al solito sottraendo **4** alla riga di output contenuta in **DH**. L'offset dell'elemento viene letto da **DI**; il codice **ASCII** dell'elemento viene letto conseguentemente da **[DI]** e viene trasferito in **AL**. Questo stesso codice **ASCII**

(contenuto ancora in **AL**) viene poi trasferito in **[BX]**; in questo modo si ottiene una stringa **C** formata da un codice **ASCII** e da uno zero, che puo' essere stampata da **writeString**. Per passare all'elemento successivo procediamo poi all'aggiornamento del puntatore **DI**; come si puo' notare, **DI** viene incrementato di un byte per volta in quanto stiamo operando su vettori di byte.

Oltre ai vari linguaggi di programmazione, anche i **SO** come il **DOS**, **Windows**, **Unix/Linux**, etc, offrono il pieno supporto delle stringhe **ASCII**; questo supporto consiste in una serie di procedure per le stringhe che i **SO** mettono a disposizione degli utenti. In un precedente capitolo abbiamo visto ad esempio che attraverso il servizio n. **09h** dell'**INT 21h** e' possibile chiamare una **ISR** del **DOS** che visualizza una stringa **ASCII** sullo schermo; questa **ISR** si aspetta che la stringa da visualizzare termini con il codice **ASCII** del simbolo **'$'**. In ambiente **Unix** i vari servizi di sistema che lavorano con le stringhe, si aspettano che le stringhe stesse terminino con un byte di valore zero; come si puo' facilmente intuire, tutti questi accorgimenti hanno lo scopo fondamentale di stabilire una serie di regole che rendano piu' semplice e veloce la manipolazione delle stringhe **ASCII**. Proprio per questo motivo in campo informatico esistono diverse convenzioni per la definizione delle stringhe **ASCII**; le due convenzioni piu' importanti sono la **convenzione C** e la **convenzione Pascal**.

Secondo la convenzione del linguaggio **C**, una stringa viene definita come un vettore di byte terminato da uno zero; in questo caso si parla di **stringa C** o **zero terminated string**. Consideriamo la seguente definizione della stringa **strByte** in versione **C**:
`char strByte[] = "Stringa ASCII";`.
Quando il compilatore **C** incontra questa definizione, predispone **14** locazioni di memoria consecutive e contigue, ciascuna delle quali occupa un byte; nelle prime **13** locazioni vengono caricati i codici **ASCII** dei **13** simboli visibili nella definizione della stringa (esattamente come si vede in Figura 3), mentre nella quattordicesima locazione viene caricato il valore **00h**. Una stringa **C** quindi occupa in memoria **1** byte in piu' rispetto al numero di simboli che formano la stringa stessa. Nel linguaggio **C** gli indici dei vettori partono da zero, per cui i vari elementi sono rappresentati dalle notazioni:
strByte[0], **strByte[1]**, **strByte[2]**, etc; se vogliamo visualizzare tutti i dettagli (indice, offset, codice **ASCII** e simbolo **ASCII**) degli elementi della stringa **strByte** possiamo scrivere il seguente programma in linguaggio **C**:

```
#include < stdio.h >
#include < string.h >

char  strByte[] = "Stringa ASCII";
int   i;

int main(void)
{
   printf("strlen(strByte) = %d\n", strlen(strByte));
   printf("sizeof(strByte) = %d\n", sizeof(strByte));
   printf("strByte[13] = %d\n", strByte[13]);
   for (i = 0; strByte[i] != '\0'; i++)
      printf("%.2d  %.4X  %.2X  %c\n", i, &strByte[i], strByte[i], strByte[i]);
   return 0;
}
```

In questo esempio la stringa viene gestita in notazione vettoriale solo per scopo didattico; e' chiaro che un programmatore **C** degno di questo nome utilizzerebbe invece i puntatori. La funzione **strlen** del **C** restituisce la lunghezza "apparente" della stringa, e' cioe' **13**; la funzione **sizeof** restituisce invece la dimensione effettiva in byte dell'oggetto **strByte**, e' cioe' **14**. Il programma stampa anche il contenuto dell'ultimo elemento (**strByte[13]**) della stringa; in questo modo si puo' constatare che

strByte[13] contiene proprio il valore **0** aggiunto automaticamente dal compilatore **C** alla fine della stringa.
Il vantaggio evidente della convenzione **C** e' dato dal fatto che le stringhe possono avere una lunghezza (teoricamente) arbitraria; infatti una stringa **C** continua finche' non viene trovato lo zero finale. Lo svantaggio altrettanto evidente e' dato dal fatto che le stringhe **C** vengono gestite in modo relativamente lento a causa della continua necessita' di testare la condizione di fine stringa; per verificare in dettaglio questi aspetti dobbiamo passare all'Assembly. Prima di tutto definiamo la stringa **strByte** in versione **C**:
`strByte db 'Stringa ASCII', 0.`
A questo punto possiamo procedere con la scansione completa della stringa; visto che la condizione di uscita dal loop e' rappresentata dal raggiungimento della fine della stringa, possiamo scrivere:

```
   mov      bx, offset strByte   ; bx punta a strByte
stringaC_loop:                   ; inizio del loop
   mov      al, [bx]             ; elemento da esaminare
   test     al, al               ; fine stringa ?
   jz       short exit_loop      ; se al = 0 esci dal loop
   ......................        ; altre istruzioni
   inc      bx                   ; aggiornamento puntatore
   jmp      short stringaC_loop  ; controllo del loop
exit_loop:
```

Come si puo' notare, abbiamo rinunciato all'istruzione **LOOP** che viene sostituita da un salto incondizionato; l'aspetto fondamentale da sottolineare riguarda il test di fine stringa che deve essere effettuato ad ogni iterazione. Conviene inserire questo test all'inizio del loop in modo da gestire anche il caso di stringhe **C** di lunghezza nulla (formate solo dallo zero finale); e' evidente che il test da effettuare ad ogni iterazione provoca un sensibile rallentamento del programma. Per le stringhe definite staticamente, possiamo determinare la relativa lunghezza in fase di assemblaggio del programma; questo sistema ovviamente non puo' essere applicato alle stringhe definite dinamicamente in fase di esecuzione del programma. In un caso del genere la lunghezza di una stringa deve essere determinata con un apposito loop contenente il solito test di fine stringa; un accorgimento importante che possiamo utilizzare in Assembly per velocizzare l'algoritmo appena illustrato consiste nell'associare il salto verso **exit_loop** alla condizione piu' improbabile (cioe' **AL=0**).

Secondo la convenzione del linguaggio **Pascal**, una stringa viene definita come un vettore di byte il cui primo elemento contiene la lunghezza della stringa stessa; in questo caso si parla di **stringa Pascal**. Consideriamo la seguente definizione della stringa **strByte** in versione **Pascal**:
`var strByte: string[13];.`
Quando il compilatore **Pascal** incontra questa definizione, predispone **14** locazioni di memoria consecutive e contigue, ciascuna delle quali occupa un byte; nella prima locazione viene caricato il valore **13d** che rappresenta la lunghezza massima consentita per la stringa. Anche una stringa **Pascal** quindi occupa in memoria **1** byte in piu' rispetto al numero di simboli che formano la stringa stessa. Nel linguaggio **Pascal** gli indici degli elementi di una stringa partono da uno, per cui i vari elementi sono rappresentati dalle notazioni:
strByte[1], **strByte[2]**, **strByte[3]**, etc; in realta' esiste anche l'elemento **strByte[0]** che come abbiamo appena visto, contiene la lunghezza della stringa. Tenendo conto del fatto che con un byte possiamo esprimere tutti i numeri senza segno che vanno da **0** a **255**, possiamo dire che la massima lunghezza consentita per una stringa **Pascal** e' di **255** elementi; in effetti questo e' proprio il difetto fondamentale della convenzione **Pascal** per le stringhe. La stringa **strByte** puo' essere definita anche come:
`var strByte: string;`

In questo caso il compilatore **Pascal** riserva a **strByte 256** locazioni di memoria consecutive e contigue, ciascuna delle quali occupa un byte, con il primo byte che contiene il valore **255**.
Il vantaggio offerto dalle stringhe **Pascal** e' dato dal fatto che conoscendo preventivamente la loro lunghezza, possiamo manipolarle in modo relativamente semplice e rapido; se vogliamo visualizzare tutti i dettagli (indice, offset, codice **ASCII** e simbolo **ASCII**) degli elementi della stringa **strByte** possiamo scrivere il seguente programma **Pascal**:

```
program Stringhe

var   strByte:    string[13];
      i:          Integer;

begin
   strByte:= 'Stringa ASCII';
   WriteLn('Length(strByte) = ', Length(strByte));
   WriteLn('SizeOf(strByte) = ', SizeOf(strByte));
   WriteLn('strByte[0] = ', Ord(strByte[0]));

   for i:= 1 to Ord(strByte[0]) do
      WriteLn(i:8, Ofs(strByte[i]):8, Ord(strByte[i]):8, strByte[i]:8);
end.
```

La funzione **Length** del **Pascal** restituisce la lunghezza "apparente" della stringa, e' cioe' **13**; la funzione **SizeOf** restituisce invece la dimensione effettiva in byte dell'oggetto **strByte**, e' cioe' **14**.
La funzione **Ord** restituisce il contenuto numerico di **strByte[0]**, e' cioe' **13**; per velocizzare il ciclo **FOR** possiamo utilizzare proprio **Ord(strByte[0])** che rappresenta la condizione di uscita dal ciclo stesso (**13** loop a partire dall'indice **1**).
Questi aspetti risultano ancora piu' evidenti in Assembly; prima di tutto definiamo la stringa **strByte** in versione **Pascal**:

```
strByte db 13, 'Stringa ASCII'
```

(come si puo' notare, il primo elemento della stringa contiene la lunghezza **13** della stringa stessa).
A questo punto possiamo procedere con la scansione completa di **strByte**; questa volta grazie al fatto che conosciamo gia' la lunghezza della stringa, possiamo sfruttare l'istruzione **LOOP**:

```
   mov      bx, offset strByte   ; bx punta a strByte
   xor      ch, ch               ; ch = 0
   mov      cl, [bx]             ; contatore = strByte[0]
   inc      bx                   ; salta strByte[0]
stringaPas_loop:                 ; inizio del loop
   ........................      ; altre istruzioni
   inc      bx                   ; aggiornamento puntatore
   loop     stringaPas_loop      ; controllo del loop
```

Risulta abbastanza evidente che l'assenza del test di fine stringa consente alla CPU di eseguire questo loop molto velocemente; nel resto del capitolo vengono esposti ulteriori dettagli sulle stringhe **C** e **Pascal**.

Richiami sui metodi di indirizzamento dei dati in Assembly.

Quando si utilizzano le istruzioni per le stringhe, si presenta spesso la necessita' di modificare il contenuto dei registri di segmento **DS** ed **ES** che come sappiamo referenziano i segmenti di dati di un programma; si tratta di una situazione molto delicata che richiede molta attenzione da parte del programmatore. Proprio per questo motivo, e' necessario richiamare alcuni concetti esposti nei precedenti capitoli relativamente ai metodi di indirizzamento dei dati in Assembly.
Abbiamo visto che in un programma Assembly e' possibile definire dati in qualsiasi segmento di programma, compresi i segmenti di codice e di stack; per accedere ai dati possiamo utilizzare il loro nome (**accesso per nome**) o i registri puntatori (**accesso per indirizzo**). Nel caso di accesso per nome, in assenza di altre indicazioni da parte del programmatore la CPU utilizza **DS** come registro di segmento predefinito; nel caso invece di accesso per indirizzo, in assenza di altre indicazioni da parte del programmatore la CPU associa **DS** ai registri puntatori **BX**, **SI**, **DI**, e associa **SS** ai registri puntatori **SP**, **BP**. Se vogliamo aggirare queste regole dobbiamo ricorrere al segment override; in questo caso utilizzando l'operatore **:** (due punti), dobbiamo indicare esplicitamente il registro di segmento che la CPU deve utilizzare. Naturalmente i vari registri di segmento devono essere gia' stati correttamente inizializzati; supponiamo ad esempio di avere il seguente segmento dati:

```
DATASEGM    SEGMENT PARA PUBLIC USE16 'DATA'

varWord     dw    3500
vectorWord  dw    8000, 5800, 3400, 2850, 3600, 5000

DATASEGM    ENDS
```

Se vogliamo referenziare **DATASEGM** attraverso **DS** dobbiamo scrivere le seguenti istruzioni:

```
mov      ax, DATASEGM      ; carica DATASEGM
mov      ds, ax            ; in DS attraverso AX
```

Queste due istruzioni assumono un'importanza fondamentale nella fase di esecuzione del programma; al momento di caricare il programma in memoria, il **SO** procede alla rilocazione dei vari segmenti di programma assegnando a ciascuno di essi un paragrafo di memoria. Supponiamo che al segmento **DATASEGM** venga assegnato il paragrafo **0400h**; di conseguenza, possiamo dire che il dato **varWord** si trova in memoria all'indirizzo logico **0400h:0000h**, mentre i vari elementi di **vectorWord** vengono a trovarsi in memoria agli indirizzi logici **0400h:0002h**, **0400h:0004h**, **0400h:0006h**, etc. Se ora vogliamo accedere per nome a **varWord** dobbiamo scrivere:
`mov ax, ds:varWord`; l'assembler incontrando questa istruzione genera il codice macchina per il trasferimento dati a **16** bit dall'offset **0000h** di **varWord** verso **AX**. Non viene inserito nessun segment override prefix in quanto **DS** e' il registro di segmento predefinito per l'accesso al blocco dati del programma. Quando la CPU incontra questo codice macchina, per accedere a **varWord** utilizza **DS** come registro di segmento predefinito e crea l'indirizzo logico **DS:0000h** dove **0000h** e' l'offset di **varWord**; questo indirizzo logico punta a **varWord** solo se **DS** e' stato correttamente inizializzato con il paragrafo **0400h** assegnato a **DATASEGM** dal **SO**. Per questo motivo, prima di accedere a qualunque dato definito in **DATASEGM** e' importantissimo inizializzare **DS** con le istruzioni mostrate in precedenza; questo discorso vale naturalmente per qualsiasi altro registro di segmento usato per accedere ai dati del programma. Se vogliamo referenziare **DATASEGM** con il registro **ES**, prima di accedere a qualunque dato definito in **DATASEGM** dobbiamo scrivere:

```
mov      ax, DATASEGM      ; carica DATASEGM
mov      es, ax            ; in ES attraverso AX
```

In fase di esecuzione del programma queste due istruzioni caricano in **ES** il paragrafo che il **SO** ha assegnato a **DATASEGM**; in questo modo tutti gli indirizzi logici riferiti a **ES** verranno gestiti correttamente dalla CPU. Siccome la CPU utilizza **DS** come registro di segmento predefinito, se vogliamo usare **ES** dobbiamo scrivere:
`mov ax, es:varWord`. In questo caso l'assembler inserisce nel codice macchina di questa istruzione il segment override prefix relativo a **ES**; la CPU incontrando questo codice macchina accede quindi a **varWord** attraverso **ES:0000h**. Analogamente, se **varWord** viene definita in un blocco codice referenziato da **CS** dobbiamo scrivere:
`mov ax, cs:varWord`; in questo modo permettiamo all'assembler di inserire nel codice macchina il segment override prefix relativo a **CS**. Per evitare il fastidio di dover specificare ogni volta il registro di segmento, possiamo delegare questo compito all'assembler attraverso la direttiva **ASSUME**; se ad esempio vogliamo referenziare **DATASEGM** con **ES** possiamo usare la direttiva:
`assume es: DATASEGM` che associa **DATASEGM** a **ES**.
In questo caso possiamo scrivere tranquillamente l'istruzione:
`mov ax, varWord`; grazie alla direttiva **ASSUME**, quando l'assembler incontra questa istruzione provvede automaticamente ad inserire nel codice macchina il segment override prefix relativo a **ES**.

La direttiva **ASSUME** diventa inutile nel caso di accesso ai dati tramite i puntatori; in questo caso, come e' stato appena detto, la CPU utilizza regole predefinite che possiamo aggirare attraverso il segment override. Naturalmente, il concetto fondamentale da ribadire ancora una volta riguarda il fatto che i registri di segmento impiegati per i segment override devono essere gia' stati correttamente inizializzati.
Si osservi che **ASSUME** non serve neanche nel caso di istruzioni che non richiedono indirizzamenti come ad esempio un trasferimento dati da **imm** a **reg**; un caso del genere e' rappresentato dall'istruzione:
`mov ax, offset varWord` che trasferisce in **AX** un valore immediato rappresentato dalla distanza in byte (offset) che separa **varWord** dall'inizio del blocco **DATASEGM** (dove il dato e' stato definito).

A questo punto siamo in grado di affrontare anche i casi piu' complessi che richiedono la necessita' di modificare il contenuto di un registro di segmento come **DS** o **ES**; naturalmente si deve evitare nella maniera piu' assoluta di modificare il contenuto di **CS** o di **SS**.
Supponiamo di voler accedere prima ad una variabile chiamata **var1** definita in un blocco **DATASEGM**, poi ad una variabile chiamata **var2** definita in un blocco **DATASEGM2** e infine ad una variabile chiamata **var3** definita anch'essa in **DATASEGM**; vogliamo inoltre utilizzare **DS** per referenziare entrambi i blocchi di programma. Rinunciando alle direttive **ASSUME** dobbiamo scrivere:

```
mov      ax, DATASEGM      ; carica DATASEGM
mov      ds, ax            ; in ds
mov      bx, ds:var1       ; accede a var1

mov      ax, DATASEGM2     ; carica DATASEGM2
mov      ds, ax            ; in ds
mov      cx, ds:var2       ; accede a var2

mov      ax, DATASEGM      ; carica DATASEGM
mov      ds, ax            ; in ds
mov      dx, ds:var3       ; accede a var3
```

I manuali del **Borland TASM** definiscono pessimo questo stile di programmazione che ci costringe ad inserire segment overrides dappertutto; in effetti la situazione diventa piutosto pesante nel caso in cui si debba accedere numerose volte ai dati definiti in un determinato segmento di programma. Per evitare di dover gestire direttamente tutti i segment overrides possiamo delegare questo compito all'assembler scrivendo:

```
mov      ax, DATASEGM       ; carica DATASEGM
mov      ds, ax             ; in ds
assume   ds: DATASEGM       ; associa DATASEGM a ds
mov      bx, var1           ; accede a var1

mov      ax, DATASEGM2      ; carica DATASEGM2
mov      ds, ax             ; in ds
assume   ds: DATASEGM2      ; associa DATASEGM2 a ds
mov      cx, var2           ; accede a var2

mov      ax, DATASEGM       ; carica DATASEGM
mov      ds, ax             ; in ds
assume   ds: DATASEGM       ; associa DATASEGM a ds
mov      dx, var3           ; accede a var3
```

Un'altra situazione interessante si presenta quando ad esempio dobbiamo trasferire il contenuto di una variabile nel contenuto di un'altra variabile, con le due variabili che si trovano in due segmenti di programma differenti; utilizziamo ad esempio le due variabili **var1** e **var2** definite rispettivamente nei due blocchi **DATASEGM** e **DATASEGM2** dell'esempio precedente. In questo caso possiamo sfruttare i due registri di segmento **DS** ed **ES**; la cosa migliore da fare consiste nell'associare **DS** al segmento dati principale del programma e **ES** all'altro segmento dati. Utilizzando **DATASEGM** come segmento dati principale e volendo copiare **var1** in **var2** possiamo scrivere:

```
mov      ax, DATASEGM       ; carica DATASEGM
mov      ds, ax             ; in ds
assume   ds: DATASEGM       ; associa DATASEGM a ds

mov      ax, DATASEGM2      ; carica DATASEGM2
mov      es, ax             ; in ds
assume   es: DATASEGM2      ; associa DATASEGM2 a es

mov      ax, var1           ; ax = var1
mov      var2, ax           ; var2 = ax = var1
```

Volendo rinunciare alle direttive **ASSUME** dobbiamo riscrivere le ultime due istruzioni in questo modo:

```
mov      ax, ds:var1        ; ax = var1
mov      es:var2, ax        ; var2 = ax = var1
```

Nel caso dei linguaggi di programmazione di alto livello, tutti questi aspetti vengono gestiti direttamente dal compilatore o dall'interprete del linguaggio stesso; nel caso dell'Assembly invece, tutto questo lavoro e' a carico del programmatore. Come si puo' facilmente immaginare, si tratta di concetti veramente importanti che devono essere gestiti con la massima cautela per evitare errori subdoli difficilissimi da scovare; in ogni caso, come abbiamo appena visto basta seguire la logica e il buon senso per riuscire a destreggiarsi anche nelle situazioni piu' complicate. Del resto la programmazione in Assembly si rivolge proprio a chi vuole affrontare situazioni di questo genere; non bisogna dimenticare tra l'altro che i puntatori giocano spesso brutti scherzi anche ai progettisti

dei compilatori, degli interpreti e degli assembler.

A questo punto iniziamo ad esaminare in dettaglio le istruzioni che le CPU **80x86** mettono a disposizione per le stringhe; come e' stato detto in precedenza queste istruzioni operano sulle stringhe di byte, di word e di dword (solo CPU **80386** e superiori). Tutte le istruzioni per le stringhe utilizzano due operandi che possiamo chiamare come al solito **SRC** e **DEST**; agli operandi di tipo **mem** si deve accedere obbligatoriamente attraverso i registri puntatori **SI** e **DI**.

Se **SRC** e' di tipo **mem**, per accedere all'operando si utilizza obbligatoriamente **SI** che proprio per questo motivo viene chiamato **Source Index Register**; in assenza del segment override il registro di segmento predefinito e' **DS** (che quindi puo' anche essere omesso), mentre in presenza del segment override e' possibile utilizzare qualsiasi registro di segmento tra **CS**, **SS**, **DS**, **ES** (piu' **FS** e **GS** per le CPU **80386** e superiori).

Se **DEST** e' di tipo **mem**, per accedere all'operando si utilizza obbligatoriamente **DI** che proprio per questo motivo viene chiamato **Destination Index Register**; il registro di segmento predefinito e' **ES** (che quindi puo' anche essere omesso) ed e' proibita qualsiasi altra forma di segment override.

Le istruzioni STOS, STOSB, STOSW, STOSD.

Con il mnemonico **STOS** si indica l'istruzione **STOre String data** (accesso in scrittura ad una stringa); questa istruzione richiede due operandi che svolgono il ruolo di **SRC** e **DEST**. Il programmatore deve specificare solo **DEST** che deve essere una locazione di memoria da **8/16/32** bit, mentre **SRC** e' implicitamente **AL/AX/EAX**; in base all'ampiezza in bit di **DEST** la CPU decide se utilizzare **AL**, **AX**, o **EAX**. Per accedere a **DEST** la CPU utilizza obbligatoriamente la coppia **ES:DI**; e' chiaro quindi che il programmatore deve preventivamente far puntare **ES:DI** a **DEST**. In assenza di segment override la CPU utilizza ugualmente **ES** come registro di segmento predefinito; e' proibita qualsiasi altra forma di segment override. Il codice macchina di **STOS** e' formato dall'unico opcode **1010101_w**; quando la CPU incontra questo codice macchina legge il contenuto di **AL/AX/EAX** e lo trasferisce nella locazione di memoria da **1/2/4** byte puntata da **ES:DI**. Subito dopo la CPU aggiorna il puntatore **DI** in base al contenuto del flag **DF** (Direction Flag); se **DF=0** il puntatore **DI** viene incrementato di **1/2/4** byte, mentre se **DF=1** il puntatore **DI** viene decrementato di **1/2/4** byte. Per specificare la dimensione in byte del dato da trasferire bisogna utilizzare il type override; vediamo alcuni esempi che chiariscono questi concetti.
Quando si utilizza l'istruzione **STOS**, l'aspetto piu' importante da gestire riguarda la preventiva inizializzazione dei registri **ES** e **DI**; il registro **DI** deve contenere l'offset di **DEST**, mentre **ES** deve referenziare il segmento di programma in cui si trova **DEST**. Il valore da trasferire in **DEST** deve essere caricato in **AL/AX/EAX** a seconda dell'ampiezza in bit del trasferimento dati; se necessario bisogna usare le istruzioni **CLD** e **STD** per selezionare attraverso il flag **DF** il tipo di aggiornamento da effettuare su **DI**.
Per analizzare in pratica il funzionamento di **STOS**, supponiamo di avere un segmento di programma **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** e' presente una variabile a **8** bit chiamata **varByte**. Se vogliamo usare **STOS** per caricare il valore **5Bh** in **varByte** possiamo scrivere le seguenti istruzioni:

```
mov     ax, ds                ; copia ds
mov     es, ax                ; in es
mov     di, offset varByte    ; es:di punta a varByte
mov     al, 5Bh               ; operando sorgente
stos    byte ptr es:[di]      ; mov es:[di], al
```

Come si puo' notare, le prime due istruzioni copiano in **ES** il contenuto di **DS** in modo che l'istruzione **STOS** possa operare correttamente; la direttiva **ASSUME** che associa **ES** a **DATASEGM** non serve in quanto **STOS** accede a **DEST** attraverso i puntatori. Osserviamo che **ES** e' il registro di segmento predefinito per **STOS** per cui lo possiamo anche omettere; questo significa che anche se scriviamo:
`stos byte ptr [di]`, la CPU utilizzera' ugualmente **ES** per formare l'indirizzo completo **ES:DI**.
Subito dopo l'esecuzione di **STOS** la CPU aggiorna il puntatore **DI** in base al contenuto di **DF**; trattandosi di un trasferimento dati a **8** bit (**1** byte) se **DF=0** il puntatore **DI** viene incrementato di **1** byte, mentre se **DF=1** il puntatore **DI** viene decrementato di **1** byte. Nel caso del nostro esempio l'aggiornamento di **DI** non ha nessuna importanza in quanto stiamo semplicemente trasferendo un valore numerico in una variabile; l'aggiornamento di **DI** assume invece una notevole importanza quando **STOS** viene utilizzata all'interno di un loop come viene mostrato piu' avanti.
In sostanza, quando si usa **STOS** bisogna sempre ricordarsi di caricare preventivamente in **ES** il segmento di programma in cui si trova **DEST**; se ad esempio la variabile **varByte** fosse stata definita in un segmento di codice referenziato da **CS**, prima di usare **STOS** avremmo dovuto copiare **CS** in **ES**.
In base alle considerazioni appena esposte, si puo' dire che la riga:
`stos byte ptr es:[di]`, e' equivalente alla riga:
`mov es:[di], al` che pero' non aggiorna **DI**.

L'esempio appena illustrato non evidenzia le potenzialita' dell'istruzione **STOS**; si puo' dire anzi che in un caso del genere e' preferibile utilizzare **MOV** che si rivela molto piu' veloce ed efficiente di **STOS**. La situazione cambia radicalmente nel momento in cui si ha la necessita' di operare su blocchi di memoria di grosse dimensioni; in questo caso **STOS** diventa molto piu' conveniente di **MOV**.
Usualmente **STOS** viene utilizzata per inizializzare o reinizializzare i dati contenuti in un blocco di memoria; attraverso **STOS**, questo blocco di memoria puo' essere gestito come una stringa di byte, di word o di dword. Analizzando la tabella delle istruzioni della CPU si puo' notare che il numero di cicli di clock necessari alla CPU per eseguire **STOS** e' sempre lo stesso indipendentemente dall'ampiezza in bit (**8**, **16** o **32**) del dato da trasferire; e' chiaro quindi che quando e' possibile conviene sempre utilizzare **STOS** con operandi a **32** bit, e questo vale per tutte le istruzioni illustrate in questo capitolo.
Per chiarire questi aspetti supponiamo di avere un blocco dati **DATASEGM** referenziato da **DS** e contenente il seguente vettore di **8000** byte:
`vectorByte db 8000 dup (0).`
Ad un certo punto del nostro programma, **vectorByte** si e' riempito di informazioni che non ci servono piu' e che dobbiamo cancellare; in pratica, vogliamo riazzerare il contenuto di tutti gli elementi di **vectorByte**. Utilizzando **STOS** in combinazione con **LOOP** possiamo scrivere:

```
VECTOR_SIZE = 8000                 ; dimensione in byte del vettore

   mov      ax, ds                 ; copia il contenuto di ds
   mov      es, ax                 ; in es attraverso ax
   mov      di, offset vectorByte  ; es:di punta a vectorByte
   xor      al, al                 ; SRC = al = 0
   mov      cx, VECTOR_SIZE        ; contatore = VECTOR_SIZE
   cld                             ; df = 0 (incremento)
stos_loop:                         ; inizio del loop
   stos     byte ptr es:[di]       ; mov es:[di], al
   loop     stos_loop              ; controllo del loop
```

Il loop viene reso piu' semplice dal fatto che l'aggiornamento del puntatore **DI** viene gestito direttamente da **STOS**; come si puo' notare, prima del loop l'istruzione **CLD** (Clear Direction Flag)

pone **DF=0** in modo che **DI** venga incrementato. Siccome stiamo effettuando un trasferimento dati da **1** byte, **DI** verra' incrementato di **1** byte alla volta; inoltre la CPU utilizza implicitamente **AL** come **SRC**. Per generare il codice macchina di **STOS** l'assembler pone **w=0** ottenendo **10101010b = AAh**.

Osservando che **vectorByte** ha una lunghezza in byte divisibile per due, possiamo velocizzare il loop trasferendo una word alla volta anziche' un byte alla volta; il loop viene quindi riscritto in questo modo:

```
VECTOR_SIZE = 4000                  ; dimensione in word del vettore

   mov      ax, ds                  ; copia il contenuto di ds
   mov      es, ax                  ; in es attraverso ax
   mov      di, offset vectorByte   ; es:di punta a vectorByte
   xor      ax, ax                  ; SRC = ax = 0
   mov      cx, VECTOR_SIZE         ; contatore = VECTOR_SIZE
   cld                              ; df = 0 (incremento)
stos_loop:                          ; inizio del loop
   stos     word ptr es:[di]        ; mov es:[di], ax
   loop     stos_loop               ; controllo del loop
```

In questo caso, dopo ogni trasferimento dati a **16** bit, **DI** viene incrementato di **2** byte; come **SRC** la CPU utilizza implicitamente il registro **AX** che e' stato inizializzato con il valore **0**. In base alle considerazioni svolte in precedenza possiamo dire che usando **STOS** con operandi a **16** bit, il tempo di esecuzione del loop viene dimezzato rispetto al caso di **STOS** con operandi a **8** bit; osserviamo infatti che in questo caso **STOS** nello stesso intervallo di tempo riesce a trasferire il doppio delle informazioni. Per generare il codice macchina di **STOS** l'assembler pone **w=1** ottenendo **10101011b = ABh**.

Osservando che **vectorByte** ha una lunghezza in byte divisibile per quattro, possiamo ulteriormente velocizzare il loop trasferendo una dword alla volta anziche' una word alla volta; naturalmente per fare questo dobbiamo disporre di una CPU **80386** o superiore in modo da poter abilitare il set di istruzioni a **32** bit. In questo caso possiamo riscrivere il loop in questo modo:

```
VECTOR_SIZE = 2000                  ; dimensione in dword del vettore

   mov      ax, ds                  ; copia il contenuto di ds
   mov      es, ax                  ; in es attraverso ax
   mov      di, offset vectorByte   ; es:di punta a vectorByte
   xor      eax, eax                ; SRC = eax = 0
   mov      cx, VECTOR_SIZE         ; contatore = VECTOR_SIZE
   cld                              ; df = 0 (incremento)
stos_loop:                          ; inizio del loop
   stos     dword ptr es:[di]       ; mov es:[di], eax
   loop     stos_loop               ; controllo del loop
```

In questo caso, dopo ogni trasferimento dati a **32** bit, **DI** viene incrementato di **4** byte; come operando sorgente la CPU utilizza implicitamente il registro **EAX** che e' stato inizializzato con il valore **0**. Con il trasferimento dati a **32** bit otteniamo un ulteriore dimezzamento del tempo di esecuzione rispetto al trasferimento dati a **16** bit; per generare il codice macchina di **STOS** l'assembler usa l'operand size prefix **66h** e pone **w=1** ottenendo **66h**, **10101011b = ABh**.

I tre esempi appena illustrati rappresentano i tre casi fondamentali per l'istruzione **STOS**; proprio in relazione a questi tre casi la **Intel** ha previsto i tre mnemonici **STOSB**, **STOSW** e **STOSD**. Queste tre istruzioni non richiedono nessun operando in quanto utilizzano implicitamente **AL/AX/EAX** come **SRC** e
ES:[DI] come **DEST**; in questo modo si evita al programmatore il fastidio di dover specificare ogni volta l'operando destinazione che del resto e' sempre **ES:[DI]**. Come si intuisce dai mnemonici, queste tre istruzioni eseguono il trasferimento dati rispettivamente a **8**, **16** e **32** bit; possiamo dire quindi che:

STOSB = STOre String Byte: esegue `mov es:[di], al` e aggiorna **DI** di **1** byte.
STOSW = STOre String Word: esegue `mov es:[di], ax` e aggiorna **DI** di **2** byte.
STOSD = STOre String Dword: esegue `mov es:[di], eax` e aggiorna **DI** di **4** byte.

I codici macchina di queste tre istruzioni sono ovviamente gli stessi dei tre casi previsti per **STOS**; per fare un esempio, l'istruzione:
`stos word ptr es:[di]`, puo' essere sostituita dall'istruzione:
`stosw`.
Quando si impiegano le istruzioni **STOSB**, **STOSW** e **STOSD** la CPU utilizza automaticamente gli operandi impliciti descritti in precedenza; il programmatore non puo' alterare in alcun modo questa situazione. Lo stesso discorso vale per l'istruzione **STOS** che utilizza obbligatoriamente l'accumulatore come **SRC** e **ES:[DI]** come **DEST**; e' proibita quindi ogni altra forma di segment override. Se ad esempio si prova a scrivere:
`stos ds:[di]`, si ottiene un messaggio di errore generato dall'assembler; l'esecuzione delle istruzioni **STOS**, **STOSB**, **STOSW** e **STOSD** non modifica nessun campo del **Flags Register**.

Le istruzioni <u>LODS</u>, <u>LODSB</u>, <u>LODSW</u>, <u>LODSD</u>.

Con il mnemonico **LODS** si indica l'istruzione **LOaD String data** (accesso in lettura ad una stringa); questa istruzione richiede due operandi che svolgono il ruolo di **SRC** e **DEST**. Il programmatore deve specificare solo **SRC** che deve essere una locazione di memoria da **8/16/32** bit, mentre **DEST** e' implicitamente **AL/AX/EAX**; in base all'ampiezza in bit di **SRC** la CPU decide se utilizzare **AL**, **AX**, o **EAX**. Per accedere a **SRC** la CPU utilizza obbligatoriamente il registro **SI**; e' chiaro quindi che il programmatore deve preventivamente caricare in **SI** l'offset di **SRC**. In assenza di segment override la CPU utilizza **DS** come registro di segmento predefinito; e' consentito il segment override che permette di utilizzare in coppia con **SI** qualsiasi registro di segmento tra **CS**, **SS**, **DS**, **ES** (piu' **FS** e **GS** per le CPU **80386** e superiori). Il codice macchina di **LODS** e' formato dall'unico opcode **1010110_w**; quando la CPU incontra questo codice macchina legge il contenuto a **8/16/32** bit della locazione di memoria puntata da **SI** e lo trasferisce in **AL/AX/EAX**. Subito dopo la CPU aggiorna il puntatore **SI** in base al contenuto del flag **DF** (Direction Flag); se **DF=0** il puntatore **SI** viene incrementato di **1/2/4** byte, mentre se **DF=1** il puntatore **SI** viene decrementato di **1/2/4** byte. Per specificare la dimensione in byte del dato da trasferire bisogna utilizzare il type override; vediamo alcuni esempi che chiariscono questi concetti.

Il primo aspetto da osservare riguarda il fatto che **LODS** accede all'operando **SRC** attraverso la coppia predefinita **DS:SI**; generalmente il registro **DS** viene impiegato per referenziare il segmento dati principale di un programma. Nei limiti del possibile, e' meglio quindi evitare qualsiasi modifica del contenuto di **DS**; proprio per questo motivo, l'istruzione **LODS** viene incontro al programmatore permettendo di specificare un diverso registro di segmento. Il registro puntatore invece deve essere obbligatoriamente **SI**; anche se si prova a specificare un diverso registro puntatore, la CPU utilizzera' ugualmente **SI**.
La situazione piu' semplice si presenta quando **SRC** si trova in un segmento di programma

referenziato da **DS**; in questo caso per usare **LODS** non abbiamo bisogno di modificare il contenuto di **DS**. Supponiamo ad esempio di avere un segmento di programma **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** e' presente una variabile a **8** bit chiamata **varByte** e contenente il valore **3Dh**. Se vogliamo usare **LODS** per trasferire in **AL** il contenuto di **varByte** possiamo scrivere le seguenti istruzioni:

```
mov      si, offset varByte   ; ds:si punta a varByte
lods     byte ptr ds:[si]     ; mov al, ds:[si]
```

Quando l'assembler incontra l'istruzione **LODS** pone **w=0** e ottiene il codice macchina **10101100b = ACh**; quando la CPU incontra questo codice macchina, legge il contenuto a **8** bit della locazione di memoria puntata da **DS:SI** e lo trasferisce in **AL** (siccome **DS** e' il registro di segmento predefinito per **LODS** lo possiamo anche omettere). Subito dopo l'esecuzione di **LODS** la CPU aggiorna il puntatore **SI** in base al contenuto di **DF**; trattandosi di un trasferimento dati a **8** bit (**1** byte) se **DF=0** il puntatore **SI** viene incrementato di **1** byte, mentre se **DF=1** il puntatore **SI** viene decrementato di **1** byte.

Supponiamo ora che **DATASEGM** sia referenziato da **ES**; in questo caso abbiamo la possibilita' di decidere se ricorrere o meno al segment override. La scelta piu' semplice e opportuna consiste naturalmente nel ricorrere al segment override che ci permette di evitare la modifica del contenuto di **DS**; in questo caso possiamo scrivere:

```
mov      si, offset varByte   ; es:si punta a varByte
lods     byte ptr es:[si]     ; mov al, es:[si]
```

Quando l'assembler incontra l'istruzione **LODS** pone **w=0** ottenendo l'opcode **10101100b = ACh**; prima di questo opcode l'assembler inserisce il segment override prefix **26h** relativo a **ES**, ottenendo cosi' il codice macchina **26h ACh**. Quando la CPU incontra questo codice macchina legge il contenuto a **8** bit della locazione di memoria puntata da **ES:SI** e lo trasferisce in **AL**; successivamente la CPU aggiorna **SI** seguendo il procedimento gia' descritto in precedenza.

La situazione piu' delicata si presenta quando decidiamo di modificare il contenuto di **DS**; in questo caso la cosa migliore da fare consiste nel salvare il contenuto di **DS** per poterlo poi ripristinare dopo l'esecuzione di **LODS**. Supponiamo ad esempio di avere un segmento dati principale **DATASEGM** referenziato da **DS** e un segmento dati secondario **DATASEGM2** referenziato da **ES**; all'interno di **DATASEGM2** e' presente la solita variabile **varByte** a **8** bit. Se vogliamo usare **LODS** per trasferire in **AL** il contenuto di **varByte** e non abbiamo intenzione di ricorrere al segment override possiamo scrivere:

```
push     ds                   ; salva ds
mov      ax, es               ; copia es
mov      ds, ax               ; in ds
mov      si, offset varByte   ; ds:si punta a varByte
lods     byte ptr [si]        ; mov al, ds:[si]
pop      ds                   ; ripristina ds
```

Alternativamente invece di usare **PUSH** e **POP**, subito dopo l'istruzione **LODS** possiamo scrivere:

```
mov      ax, DATASEGM         ; carica DATASEGM
mov      ds, ax               ; in ds
```

In assenza di queste istruzioni di ripristino, subito dopo l'esecuzione di **LODS** il registro di segmento **DS** continua a puntare a **DATASEGM2**; qualsiasi accesso ai dati presenti nel blocco **DATASEGM** produce quindi risultati privi di senso. La CPU infatti associa a **DS** (e cioe' a **DATASEGM2**) gli offset dei dati presenti in **DATASEGM**; come si puo' immaginare, questo tipo di bug e' piuttosto subdolo in quanto nella maggior parte dei casi il programma continua a funzionare comportandosi pero' in modo strano.

In base alle considerazioni precedentemente esposte, possiamo dire che la riga:
`lods byte ptr [si]`, e' equivalente alla riga:
`mov al, [si]` che pero' non aggiorna **SI**.
In effetti, nel caso di un semplice trasferimento dati a **8/16/32** bit, l'istruzione **MOV** risulta essere molto piu' veloce ed efficiente di **LODS**; l'istruzione **LODS** (come tutte le istruzioni per le stringhe) dimostra tutte le sue potenzialita' quando opera su blocchi di memoria di grosse dimensioni. Tipicamente l'istruzione **LODS** viene utilizzata all'interno di un loop per verificare ed elaborare i dati letti da un buffer; subito dopo la verifica e l'elaborazione questi dati vengono in genere inviati ad una periferica come il modem, la stampante, etc. Nei precedenti esempi abbiamo usato **LODS** per effettuare semplici trasferimenti di dati; in casi del genere l'aggiornamento che la CPU effettua su **SI** non ha nessuna importanza. Quando invece si utilizza **LODS** all'interno di un loop, e' importantissimo utilizzare preventivamente le istruzioni **CLD** e **STD** per stabilire se **SI** debba essere incrementato o decrementato; vediamo un esempio pratico. Supponiamo di aver definito in un blocco dati referenziato da **DS** un buffer di dati chiamato **strBuffer**, formato da **STRBUFFERLEN** word; per verificare i dati presenti in questo buffer possiamo scrivere il seguente loop:

```
   mov      si, offset strBuffer     ; ds:si punta a strBuffer
   mov      cx, STRBUFFERLEN         ; contatore = STRBUFFERLEN
   cld                               ; df = 0 (incremento)
lods_loop:                           ; inizio del ciclo
   lods     word ptr [si]            ; mov ax, ds:[si]
   ......................            ; altre istruzioni
   loop     lods_loop                ; controllo del ciclo
```

Come si puo' notare, prima di entrare nel loop abbiamo usato l'istruzione **CLD** che pone **DF=0**; ad ogni iterazione, subito dopo l'esecuzione di **LODS** la CPU incrementa **SI** di **2** byte (trasferimento dati a **16** bit) permettendoci cosi' di scandire in avanti (**forwards**) tutto il buffer di dati.

Riassumendo quindi, l'istruzione **LODS** offre la possibilita' di effettuare il trasferimento dati a **8**, **16** o **32** bit; i tre codici macchina relativi a questi tre casi sono rispettivamente **ACh**, **ADh** e **66h ADh**. In relazione a queste tre possibilita' le CPU **80x86** mettono a disposizione i tre mnemonici **LODSB**, **LODSW** e **LODSD**; queste tre istruzioni non richiedono nessun operando in quanto utilizzano implicitamente **DS:[SI]** come **SRC** e **AL/AX/EAX** come **DEST**. Come si intuisce dai mnemonici, queste tre istruzioni eseguono il trasferimento dati rispettivamente a **8**, **16** e **32** bit; possiamo dire quindi che:

LODSB = LOaD String Byte: esegue `mov al, ds:[si]` e aggiorna **SI** di **1** byte.
LODSW = LOaD String Word: esegue `mov ax, ds:[si]` e aggiorna **SI** di **2** byte.
LODSD = LOaD String Dword: esegue `mov eax, ds:[si]` e aggiorna **SI** di **4** byte.

I codici macchina di queste tre istruzioni sono ovviamente gli stessi dei tre casi previsti per **LODS**; per fare un esempio, l'istruzione:
`lods word ptr ds:[si]`, puo' essere sostituita dall'istruzione:
`lodsw`.

Quando si impiegano le istruzioni **LODSB**, **LODSW** e **LODSD** la CPU utilizza automaticamente gli operandi impliciti descritti in precedenza; il programmatore non puo' alterare in alcun modo questa situazione. Se vogliamo ricorrere al segment override dobbiamo necessariamente usare l'istruzione **LODS**; l'esecuzione delle istruzioni **LODS**, **LODSB**, **LODSW** e **LODSD** non modifica nessun campo del **Flags Register**.

Le istruzioni SCAS, SCASB, SCASW, SCASD.

Con il mnemonico **SCAS** si indica l'istruzione **SCAn String data** (comparazione con un elemento di una stringa); questa istruzione richiede due operandi che svolgono il ruolo di **SRC** e **DEST**. Il programmatore deve specificare solo **DEST** che deve essere una locazione di memoria da **8/16/32** bit, mentre **SRC** e' implicitamente **AL/AX/EAX**; in base all'ampiezza in bit di **DEST** la CPU decide se utilizzare **AL**, **AX**, o **EAX**. Per accedere a **DEST** la CPU utilizza obbligatoriamente la coppia **ES:DI**; e' chiaro quindi che il programmatore deve preventivamente far puntare **ES:DI** a **DEST**. In assenza di segment override la CPU utilizza ugualmente **ES** come registro di segmento predefinito; e' proibita qualsiasi altra forma di segment override. Il codice macchina di **SCAS** e' formato dall'unico opcode **1010111_w**; quando la CPU incontra questo codice macchina, sottrae da **AL/AX/EAX** il contenuto a **8/16/32** bit dell'operando **DEST** puntato da **ES:DI**, scarta il risultato e modifica i flags **OF**, **SF**, **ZF**, **AF**, **PF** e **CF**, esattamente come succede con l'istruzione **CMP**. Subito dopo la CPU aggiorna il puntatore **DI** in base al contenuto del flag **DF** (Direction Flag); se **DF=0** il puntatore **DI** viene incrementato di **1/2/4** byte, mentre se **DF=1** il puntatore **DI** viene decrementato di **1/2/4** byte. Per specificare la dimensione in byte degli operandi da comparare bisogna utilizzare il type override; vediamo alcuni esempi che chiariscono questi concetti.

Innanzi tutto osserviamo che per quanto riguarda gli indirizzamenti ci troviamo nella stessa situazione di **STOS**; l'operando **DEST** deve essere indirizzato obbligatoriamente attraverso **ES:DI** ed e' proibita qualsiasi altra forma di segment override. Per analizzare il principio di funzionamento di **SCAS** supponiamo di avere un segmento dati **DATASEGM** referenziato come al solito da **DS**; all'interno di **DATASEGM** e' presente la definizione di una variabile a **8** bit chiamata **varByte**. L'istruzione **SCAS** puo' essere impiegata per confrontare il contenuto di **varByte** con il contenuto del registro accumulatore (in questo caso **AL**); se ad esempio vogliamo confrontare il contenuto di **varByte** con il valore **2Dh** possiamo scrivere:

```
push     ds                    ; copia ds
pop      es                    ; in es
mov      di, offset varByte    ; es:di punta a varByte
mov      al, 2Dh               ; operando sorgente
scas     byte ptr es:[di]      ; cmp al, es:[di]
```

La prima cosa da fare consiste come al solito nel copiare **DS** in **ES**; a tale proposito possiamo anche utilizzare le istruzioni **PUSH** e **POP** che ci permettono di effettuare questa copia in modo molto sbrigativo. Successivamente carichiamo in **DI** l'offset di **varByte**; in questo modo **varByte** viene puntata dalla coppia **ES:DI**. L'operando **SRC** viene caricato in **AL** in quanto vogliamo effettuare una comparazione tra operandi a **8** bit; a questo punto possiamo chiamare **SCAS** che effettua la sottrazione:
AL - ES:[DI], scarta il risultato, modifica i flags citati in precedenza e aggiorna il puntatore **DI** di **1** byte. Subito dopo l'esecuzione di **SCAS** possiamo consultare i vari flags per conoscere il risultato della comparazione; naturalmente per semplificare il nostro lavoro possiamo anche servirci delle istruzioni **Jcond**.
In base alle considerazioni appena esposte, si puo' dire che la riga:
`scas byte ptr es:[di]`, e' equivalente alla riga:
`cmp al, es:[di]` che pero' non aggiorna **DI**.

Bisogna fare molta attenzione al fatto che esiste una importante differenza tra **SCAS** e **CMP**; l'istruzione **CMP** sottrae **SRC** da **DEST**, mentre **SCAS** fa esattamente l'opposto sottraendo **DEST** da **SRC**. Chiaramente questa differenza di comportamento puo' facilmente confondere le idee; il programmatore deve necessariamente adattarsi a questa situazione in quanto la **Intel** ha imposto questo scambio di ruoli tra **SRC** e **DEST**.

Nei casi molto semplici come quello dell'esempio precedente, conviene decisamente utilizzare l'istruzione **CMP** che si rivela molto piu' efficiente e veloce di **SCAS**; l'istruzione **SCAS** diventa invece piu' vantaggiosa di **CMP** nel caso di comparazioni da effettuare su stringhe di dimensioni medio grandi.
Tipicamente l'istruzione **SCAS** viene utilizzata all'interno di un loop per effettuare la scansione degli elementi di una stringa alla ricerca di un determinato valore numerico; in questo caso si fa puntare **ES:DI** all'inizio della stringa, si carica in **AL/AX/EAX** il valore da cercare e si avvia un loop contenente l'istruzione **SCAS**. Un caso particolarmente interessante e' rappresentato dal calcolo della lunghezza in byte di una stringa **C**; per eseguire questo calcolo dobbiamo effettuare la scansione della stringa alla ricerca di un byte che vale zero (fine stringa). Possiamo dire quindi che in questo caso l'istruzione **SCAS** si rivela molto utile ed efficace; osserviamo che quando si usa **SCAS** all'interno di un loop e' molto importante inizializzare opportunamente **DF** in modo da decidere se la stringa debba essere scandita in avanti (**forwards**) o all'indietro (**backwards**).
Vediamo un esempio pratico che utilizza un algoritmo di scansione in avanti della stringa; al termine della scansione si sottrae l'offset iniziale della stringa dal contenuto finale di **DI** ottenendo in questo modo la lunghezza in byte della stringa stessa.
Supponiamo di avere un segmento dati **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** e' presente la definizione della seguente stringa **C**:

```
stringaC db 'Stringa da esaminare', 0.
```

La lunghezza di una stringa **ASCII** e' rappresentata esclusivamente dal numero di simboli **ASCII** presenti nella stringa stessa; nel caso di **stringaC** possiamo dire quindi che la lunghezza e' pari a **20** byte. A questo punto possiamo procedere con la scrittura dell'algoritmo:

```
   push     ds                     ; copia ds
   pop      es                     ; in es
   mov      di, offset stringaC    ; es:di punta a stringaC
   xor      al, al                 ; al = 0 (valore da cercare)
   cld                             ; df = 0 (incremento)
strlen_loop:                       ; inizio del ciclo
   scas     byte ptr es:[di]       ; cmp al, es:[di]
   jnz      short strlen_loop      ; se al e' diverso da es:[di] ripeti
   dec      di                     ; elimina l'incremento finale
   sub      di, offset stringaC    ; di = lunghezza stringa C
```

Come si puo' notare, prima di entrare nel loop abbiamo usato l'istruzione **CLD** che pone **DF=0**; ad ogni iterazione, subito dopo l'esecuzione di **SCAS** la CPU incrementa **DI** di **1** byte (comparazione a **8** bit) permettendoci cosi' di scandire in avanti (**forwards**) tutta la stringa. Al posto di **LOOP** utilizziamo una istruzione **Jcond** in quanto la condizione di uscita dal loop e' rappresentata dal raggiungimento della fine della stringa e non da **CX=0**; e' chiaro che utilizzando questo algoritmo con stringhe prive dello zero finale si ottengono risultati privi di senso. Alla fine del loop **DI** viene decrementato di **1** byte per eliminare l'ultimo incremento; in questo modo **DI** si trova a puntare all'offset dello zero finale della stringa. Sottraendo ora da **DI** l'offset iniziale della stringa si ottiene chiaramente la lunghezza di **stringaC**, e cioe' la distanza in byte che esiste tra lo zero finale e il primo carattere della stringa stessa; come si puo' facilmente verificare, questo algoritmo si comporta in modo corretto anche con le stringhe **C** di lunghezza nulla, formate cioe' solo dallo zero finale.

In definitiva, l'istruzione **SCAS** offre la possibilita' di effettuare comparazioni tra operandi a **8**, **16** o **32** bit; i tre codici macchina relativi a questi tre casi sono rispettivamente **AEh**, **AFh** e **66h AFh**. In relazione a queste tre possibilita' le CPU **80x86** mettono a disposizione i tre mnemonici **SCASB**, **SCASW** e **SCASD**; queste tre istruzioni non richiedono nessun operando in quanto utilizzano implicitamente **ES:[DI]** come **DEST** e **AL/AX/EAX** come **SRC**. Come si intuisce dai mnemonici, queste tre istruzioni eseguono comparazioni di dati rispettivamente a **8**, **16** e **32** bit; possiamo dire quindi che:

SCASB = SCAn String Byte: esegue `cmp al, es:[di]` e aggiorna **DI** di **1** byte.
SCASW = SCAn String Word: esegue `cmp ax, es:[di]` e aggiorna **DI** di **2** byte.
SCASD = SCAn String Dword: esegue `cmp eax, es:[di]` e aggiorna **DI** di **4** byte.

I codici macchina di queste tre istruzioni sono ovviamente gli stessi dei tre casi previsti per **SCAS**; per fare un esempio, l'istruzione:
`scas byte ptr es:[di]`, puo' essere sostituita dall'istruzione:
`scasb`.
Quando si impiegano le istruzioni **SCASB**, **SCASW** e **SCASD** la CPU utilizza automaticamente gli operandi impliciti descritti in precedenza; il programmatore non puo' alterare in alcun modo questa situazione. Lo stesso discorso vale per l'istruzione **SCAS** che utilizza obbligatoriamente l'accumulatore come **SRC** e **ES:[DI]** come **DEST**; e' proibita quindi ogni altra forma di segment override. Come abbiamo gia' visto in precedenza, l'esecuzione delle istruzioni **SCAS**, **SCASB**, **SCASW** e **SCASD** modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** del **Flags Register**.

Le istruzioni MOVS, MOVSB, MOVSW, MOVSD.

Con il mnemonico **MOVS** si indica l'istruzione **MOVe data from String to string** (trasferimento dati da stringa a stringa); questa istruzione richiede due operandi che svolgono il ruolo di **SRC** e **DEST**. Il programmatore deve specificare sia **SRC** che **DEST**; entrambi gli operandi devono essere di tipo **mem** con ampiezza **8/16/32** bit.
Per accedere a **SRC** la CPU utilizza obbligatoriamente il registro **SI**; il programmatore deve quindi caricare preventivamente in **SI** l'offset di **SRC**. In assenza di segment override la CPU utilizza **DS** come registro di segmento predefinito; e' consentito il segment override che permette di utilizzare in coppia con **SI** qualsiasi registro di segmento tra **CS**, **SS**, **DS**, **ES** (piu' **FS** e **GS** per le CPU **80386** e superiori).
Per accedere a **DEST** la CPU utilizza obbligatoriamente la coppia **ES:DI**; il programmatore deve quindi far puntare preventivamente la coppia **ES:DI** a **DEST**. In assenza di segment override la CPU utilizza ugualmente **ES** come registro di segmento predefinito; e' proibita qualsiasi altra forma di segment override.
Il codice macchina di **MOVS** e' formato dall'unico opcode **1010010_w**; quando la CPU incontra questo codice macchina legge il contenuto a **8/16/32** bit della locazione di memoria puntata da **SI** e lo trasferisce nella locazione di memoria da **8/16/32** bit puntata da **ES:DI**. Subito dopo la CPU aggiorna i puntatori **SI** e **DI** in base al contenuto del flag **DF** (Direction Flag); se **DF=0** i puntatori **SI** e **DI** vengono entrambi incrementati di **1/2/4** byte, mentre se **DF=1** i puntatori **SI** e **DI** vengono entrambi decrementati di **1/2/4** byte. Per specificare la dimensione in byte del dato da trasferire bisogna utilizzare il type override; vediamo alcuni esempi che chiariscono il funzionamento di **MOVS**.

Il primo aspetto da osservare riguarda il fatto che in relazione agli indirizzamenti dobbiamo unire assieme le cose dette per **LODS** e per **STOS**; l'operando **DEST** deve essere indirizzato obbligatoriamente attraverso **ES:DI** ed e' proibita qualsiasi altra forma di segment override, mentre l'operando **SRC** deve essere indirizzato obbligatoriamente attraverso **SI** (con registro di segmento

predefinito **DS**) ed e' consentito il segment override. Per analizzare il principio di funzionamento di **MOVS** supponiamo di avere un segmento dati **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** sono presenti le definizioni di due stringhe di byte chiamate **strSource** e **strDest**.
L'istruzione **MOVS** puo' essere impiegata per copiare il contenuto di un elemento di **strSource** in un elemento di **strDest**; se ad esempio vogliamo copiare il contenuto di **strSource[5]** in **strDest[3]** possiamo scrivere:

```
push     ds                         ; copia ds
pop      es                         ; in es (es ora referenzia DATASEGM)
mov      si, offset strSource + 5   ; ds:si punta a strSource[5]
mov      di, offset strDest + 3     ; es:di punta a strDest[3]
movs     byte ptr [di], [si]        ; mov byte ptr es:[di], ds:[si]
```

In questo esempio rinunciamo al segment override e utilizziamo quindi i due registri di segmento predefiniti che sono **DS** per **SRC** e **ES** per **DEST**; siccome **SRC** e **DEST** si trovano entrambi nello stesso blocco **DATASEGM** referenziato da **DS**, la prima cosa da fare consiste nel copiare **DS** in **ES**. Successivamente vengono inizializzati i due registri puntatori **SI** e **DI**; nel registro **SI** viene caricato l'offset dell'operando sorgente **strSource[5]**, mentre nel registro **DI** viene caricato l'offset dell'operando destinazione **strDest[3]**. A questo punto viene eseguita l'istruzione **MOVS** che esegue il trasferimento dati a **8** bit; come si puo' notare nell'esempio, i registri di segmento vengono omessi per cui la CPU associa automaticamente **SI** a **DS** e **DI** a **ES**
La situazione diventa ancora piu' semplice nel caso in cui **strSource** si trovi in un blocco dati referenziato da **DS** e **strDest** si trovi in un blocco dati referenziato da **ES**; in un caso del genere, non solo non c'e' bisogno del segment override ma non c'e' neanche bisogno di inizializzare **DS** e **ES** in quanto il loro contenuto va bene cosi' com'e'.
Il caso piu' impegnativo riguarda invece la situazione opposta che si verifica quando **strSource** si trova in un blocco dati referenziato da **ES** e **strDest** si trova in un blocco dati referenziato da **DS**; per indirizzare **DEST** e' proibito il segment override per cui siamo costretti a modificare **ES** in modo da farlo puntare al segmento dati contenente **strDest**. A questo punto si presenta il problema dell'indirizzamento di **SRC**; se disponiamo di una CPU **80386** o superiore, possiamo abilitare il set di istruzioni a **32** bit per servirci dei nuovi registri di segmento **FS** e **GS**. Utilizzando uno di questi due registri per indirizzare **SRC** evitiamo la modifica di **DS**; se invece non possiamo ricorrere a **FS** o **GS** dobbiamo necessariamente modificare anche **DS**. Come e' stato gia' detto in precedenza, l'aspetto piu' importante da ricordare riguarda il fatto che se abbiamo modificato registri di segmento che in precedenza referenziavano dei segmenti di programma, dobbiamo provvedere in seguito a ripristinare la situazione iniziale; per chiarire questi concetti molto importanti vediamo un esempio pratico.
Supponiamo che **strDest** di trovi in un blocco **DATASEGM** referenziato da **DS** e che **strSource** di trovi in un blocco **DATASEGM2** referenziato da **ES**; come per l'esempio precedente, vogliamo copiare il contenuto di **strSource[5]** in **strDest[3]**. Se abbiamo la possibilita' di utilizzare **FS** possiamo scrivere:

```
push     es                         ; copia es
pop      fs                         ; in fs (fs ora referenzia DATASEGM2)
push     ds                         ; copia ds
pop      es                         ; in es (es ora referenzia DATASEGM)

mov      si, offset strSource + 5   ; fs:si punta a strSource[5]
mov      di, offset strDest + 3     ; es:di punta a strDest[3]
movs     byte ptr es:[di], fs:[si]  ; mov byte ptr es:[di], fs:[si]

push     fs                         ; copia fs (vecchio es)
pop      es                         ; in es (es ora referenzia DATASEGM2)
```

La prima e la seconda istruzione copiano **ES** in **FS** in modo che **DATASEGM2** venga referenziato da **FS**; la terza e la quarta istruzione copiano **DS** in **ES** in modo che **DATASEGM** venga referenziato da **ES**. Per indirizzare **SRC** utilizziamo **FS:SI**, mentre per indirizzare **DEST** utilizziamo la coppia obbligatoria **ES:DI**; in questo modo evitiamo la modifica di **DS** che continua quindi a referenziare **DATASEGM**. Subito dopo l'esecuzione di **MOVS** provvediamo a ripristinare **ES** in modo che questo registro torni a referenziare **DATASEGM2**; naturalmente il vecchio contenuto di **ES** si trova in **FS**.

Se non e' possibile utilizzare **FS** o **GS**, siamo costretti a modificare sia **ES** che **DS**; in sostanza dobbiamo scambiare tra loro i contenuti di **ES** e **DS**. In questo caso dobbiamo scrivere:

```
push     ds                          ; inserisce ds nello stack
push     es                          ; inserisce es nello stack
pop      ds                          ; ds ora referenzia DATASEGM2
pop      es                          ; es ora referenzia DATASEGM

mov      si, offset strSource + 5    ; ds:si punta a strSource[5]
mov      di, offset strDest + 3      ; es:di punta a strDest[3]
movs     byte ptr [di], [si]         ; mov byte ptr es:[di], ds:[si]

push     ds                          ; inserisce ds nello stack
push     es                          ; inserisce es nello stack
pop      ds                          ; ds ora referenzia DATASEGM
pop      es                          ; es ora referenzia DATASEGM2
```

Le prime quattro istruzioni scambiano tra loro i contenuti di **ES** e **DS**; per verificarlo basta tracciare uno schema dello stack (vettore di word) e simulare le operazioni di inserimento e estrazione. Come si puo' notare l'istruzione **MOVS** viene eseguita senza ricorrere al segment override; le ultime quattro istruzioni ripristinano i registri di segmento **ES** e **DS** riportandoli alla situazione originaria.

Da tutte le considerazioni esposte, si deduce che la riga:
`movs byte ptr es:[di], ds:[si]`, puo' essere simulata con le istruzioni:

```
mov      al, ds:[si]     ; al = ds:[si]
mov      es:[di], al     ; es:[di] = al = ds:[si]
```

Apparentemente **MOVS** sembra un'istruzione per il trasferimento dati da memoria a memoria; in realta' questo trasferimento avviene attraverso i registri temporanei della CPU.
Nel caso dei semplici esempi mostrati in precedenza, conviene decisamente utilizzare l'istruzione **MOV** che si rivela nettamente piu' veloce ed efficiente di **MOVS**; l'istruzione **MOVS** diventa invece conveniente in termini di prestazioni, quando dobbiamo operare come al solito su stringhe di grosse dimensioni. Come molti avranno intuito, l'istruzione **MOVS** viene largamente utilizzata all'interno di un loop per copiare grosse quantita' di dati da una sorgente ad una destinazione; questa operazione viene resa piu' semplice grazie al fatto che l'aggiornamento dei puntatori viene gestito direttamente da **MOVS**. Per verificare le prestazioni di **MOVS** vediamo un esempio che utilizza la procedura **writeString** per visualizzare una stringa **C** che occupa tutto lo schermo; nel segmento di dati **DATASEGM** definiamo due stringhe chiamate **srcBuffer** (sorgente) e **destBuffer** (destinazione). La stringa **srcBuffer** contiene **4000** lettere **'A'** capaci di riempire completamente lo schermo in modalita' testo a **80** colonne e **50** righe
(**80 * 50 = 4000**); l'istruzione **MOVS** copia **srcBuffer** in **destBuffer**. Naturalmente **destBuffer** deve essere in grado di contenere tutti i **4000** elementi di **srcBuffer** (piu' lo zero finale); a questo punto **destBuffer** viene passata attraverso **BX** a **writeString** che provvede a visualizzarla. Prima di tutto definiamo il segmento dati del nostro programma:

```
DATASEGM    SEGMENT PARA PUBLIC USE16 'DATA'

srcBuffer   db    4000  dup  ('A')     ; stringa sorgente
destBuffer  db    4001  dup  (0)       ; 4000 byte piu' lo zero finale

DATASEGM    ENDS
```

Il blocco **DATASEGM** viene referenziato come al solito da **DS**, e siccome entrambe le stringhe si trovano in **DATASEGM**, per poter utilizzare **MOVS** dobbiamo provvedere innanzi tutto a copiare **DS** in **ES**; a questo punto possiamo scrivere:

```
   call     set80x50                    ; modalita' testo 80 * 50

SCREEN_SIZE = 4000                      ; 4000 byte da trasferire

   push     ds                          ; copia ds
   pop      es                          ; in es (es ora referenzia DATASEGM)
   mov      si, offset srcBuffer        ; ds:si punta a srcBuffer
   mov      di, offset destBuffer       ; es:di punta a destBuffer
   mov      cx, SCREEN_SIZE             ; 4000 iterazioni
   cld                                  ; df = 0 (incremento)
movs_loop:
   movs     byte ptr es:[di], ds:[si]  ; mov byte ptr es:[di], ds:[si]
   loop     movs_loop                   ; controllo del loop

   mov      bx, offset destBuffer       ; ds:bx punta a destBuffer
   xor      dx, dx                      ; riga 0, colonna 0
   call     writeString                 ; visualizza destBuffer
   call     waitChar                    ; attende la pressione di un tasto
   call     clearScreen                 ; pulisce lo schermo
   call     set80x25                    ; modalita' testo 80 * 25
```

Ad ogni iterazione, dopo l'esecuzione di **MOVS** sia **SI** che **DI** vengono incrementati di **1** byte; a tale proposito prima del loop abbiamo inserito l'istruzione **CLD** che pone **DF=0**. Osservando che **4000** e' divisibile per **4** (**4000 / 4 = 1000**), possiamo utilizzare il trasferimento dati a **32** bit che ci permette di aumentare di circa **4** volte la velocita' di esecuzione del loop; prima di tutto modifichiamo la costante **SCREEN_SIZE** scrivendo:

```
SCREEN_SIZE = 1000.
```

A questo punto possiamo modificare il loop scrivendo:

```
movs_loop:
   movs     dword ptr es:[di], ds:[si] ; mov dword ptr es:[di], ds:[si]
   loop     movs_loop                  ; controllo del loop
```

In questo caso ad ogni iterazione, dopo l'esecuzione di **MOVS** sia **SI** che **DI** vengono incrementati di **4** byte; naturalmente questo accade se abbiamo usato come al solito **CLD** che pone **DF=0**.

In definitiva, l'istruzione **MOVS** offre la possibilita' di effettuare trasferimenti di dati a **8**, **16** o **32** bit; i tre codici macchina relativi a questi tre casi sono rispettivamente **A4h**, **A5h** e **66h A5h**. In relazione a queste tre possibilita' le CPU **80x86** mettono a disposizione i tre mnemonici **MOVSB**, **MOVSW** e **MOVSD**; queste tre istruzioni non richiedono nessun operando in quanto utilizzano implicitamente **DS:[SI]** come **SRC** e **ES:[DI]** come **DEST**. Come si intuisce dai mnemonici, queste tre istruzioni eseguono il trasferimento dati rispettivamente a **8**, **16** e **32** bit; possiamo dire quindi che:

MOVSB = MOVe String Byte: esegue `mov byte ptr es:[di], ds:[si]` e aggiorna **SI** e **DI** di **1** byte.
MOVSW = MOVe String Word: esegue `mov word ptr es:[di], ds:[si]` e aggiorna **SI** e **DI** di **2** byte.
MOVSD = MOVe String Dword: esegue `mov dword ptr es:[di], ds:[si]` e aggiorna **SI** e **DI** di **4** byte.

I codici macchina di queste tre istruzioni sono ovviamente gli stessi dei tre casi previsti per **MOVS**; per fare un esempio, l'istruzione:
`movs dword ptr es:[di], ds:[si]`, puo' essere sostituita dall'istruzione:
`movsd`.
Quando si impiegano le istruzioni **MOVSB**, **MOVSW** e **MOVSD** la CPU utilizza automaticamente gli operandi impliciti descritti in precedenza; il programmatore non puo' alterare in alcun modo questa situazione. Se vogliamo ricorrere al segment override dobbiamo necessariamente usare l'istruzione **MOVS**; naturalmente il segment override e' consentito solo per il registro di segmento che referenzia **SRC**. L'esecuzione delle istruzioni **MOVS**, **MOVSB**, **MOVSW** e **MOVSD** non modifica nessun campo del **Flags Register**.

Le istruzioni <u>CMPS</u>, <u>CMPSB</u>, <u>CMPSW</u>, <u>CMPSD</u>.

Con il mnemonico **CMPS** si indica l'istruzione **CoMPare String data** (comparazione tra stringhe); questa istruzione richiede due operandi che svolgono il ruolo di **SRC** e **DEST**. Il programmatore deve specificare sia **SRC** che **DEST**; entrambi gli operandi devono essere di tipo **mem** con ampiezza **8/16/32** bit.
Per accedere a **SRC** la CPU utilizza obbligatoriamente il registro **SI**; il programmatore deve quindi caricare preventivamente in **SI** l'offset di **SRC**. In assenza di segment override la CPU utilizza **DS** come registro di segmento predefinito; e' consentito il segment override che permette di utilizzare in coppia con **SI** qualsiasi registro di segmento tra **CS**, **SS**, **DS**, **ES** (piu' **FS** e **GS** per le CPU **80386** e superiori).
Per accedere a **DEST** la CPU utilizza obbligatoriamente la coppia **ES:DI**; il programmatore deve quindi far puntare preventivamente la coppia **ES:DI** a **DEST**. In assenza di segment override la CPU utilizza ugualmente **ES** come registro di segmento predefinito; e' proibita qualsiasi altra forma di segment override.
Il codice macchina di **CMPS** e' formato dall'unico opcode **1010011_w**; quando la CPU incontra questo codice macchina legge il contenuto a **8/16/32** bit dell'operando **DEST** puntato da **ES:DI**, lo sottrae dal contenuto a **8/16/32** bit dell'operando **SRC** puntato da **SI**, scarta il risultato e modifica i flags **OF**, **SF**, **ZF**, **AF**, **PF** e **CF**, esattamente come succede con l'istruzione **CMP**. Subito dopo la CPU aggiorna i puntatori **SI** e **DI** in base al contenuto del flag **DF** (Direction Flag); se **DF=0** i puntatori **SI** e **DI** vengono entrambi incrementati di **1/2/4** byte, mentre se **DF=1** i puntatori **SI** e **DI** vengono entrambi decrementati di **1/2/4** byte. Per specificare la dimensione in byte dei dati da comparare bisogna utilizzare il type override; vediamo alcuni esempi che chiariscono il funzionamento di **CMPS**.

Prima di tutto osserviamo che in relazione agli indirizzamenti degli operandi, ci troviamo nella stessa identica situazione di **MOVS**, per cui tutte le considerazioni svolte per **MOVS** valgono anche per **CMPS**. Vediamo un semplice esempio che ci permette di capire il principio di funzionamento di **CMPS**. Supponiamo di avere un segmento dati **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** sono presenti le definizioni di due stringhe di byte chiamate **strSource** e **strDest**.

L'istruzione **CMPS** puo' essere impiegata per comparare il contenuto di un elemento di **strSource** con il contenuto di un elemento di **strDest**; se ad esempio vogliamo comparare **strSource[5]** con **strDest[3]** possiamo scrivere:

```
push     ds                          ; copia ds
pop      es                          ; in es (es ora referenzia DATASEGM)
mov      si, offset strSource + 5    ; ds:si punta a strSource[5]
mov      di, offset strDest + 3      ; es:di punta a strDest[3]
cmps     byte ptr [di], [si]         ; cmp byte ptr ds:[si], es:[di]
```

In questo esempio rinunciamo al segment override e utilizziamo quindi i due registri di segmento predefiniti che sono **DS** per **SRC** e **ES** per **DEST**; siccome **SRC** e **DEST** si trovano entrambi nello stesso blocco **DATASEGM** referenziato da **DS**, la prima cosa da fare consiste nel copiare **DS** in **ES**. Successivamente vengono inizializzati i due registri puntatori **SI** e **DI**; nel registro **SI** viene caricato l'offset dell'operando sorgente **strSource[5]**, mentre nel registro **DI** viene caricato l'offset dell'operando destinazione **strDest[3]**. A questo punto viene eseguita l'istruzione **CMPS** che esegue la comparazione tra operandi a **8** bit; come si puo' notare nell'esempio, i registri di segmento vengono omessi per cui la CPU associa automaticamente **SI** a **DS** e **DI** a **ES**. Subito dopo l'esecuzione di **CMPS** la CPU oltre ad aggiornare i puntatori, modifica anche i flags citati in precedenza; consultando questi flags possiamo avere tutte le informazioni necessarie per conoscere il risultato della comparazione.
Nel caso in cui si renda necessario il ricorso al segment override basta fare riferimento alle cose gia' dette per **MOVS**.

Da tutte le considerazioni esposte, si deduce che la riga:
`cmps byte ptr es:[di], ds:[si]`, puo' essere simulata con le istruzioni:

```
mov      al, ds:[si]     ; al = ds:[si]
cmp      al, es:[di]     ; ds:[si] - es:[di]
```

Apparentemente **CMPS** sembra un'istruzione per la comparazione tra due operandi di tipo **mem**; in realta' la comparazione avviene avviene attraverso i registri temporanei della CPU.
Cosi' come accade per **SCAS**, anche nel caso di **CMPS** bisogna fare molta attenzione al fatto che esiste una importante differenza con **CMP**; l'istruzione **CMP** sottrae **SRC** da **DEST**, mentre **CMPS** fa esattamente l'opposto sottraendo **DEST** da **SRC**. In sostanza, quando scriviamo:
`cmps es:[di], ds:[si]`, la CPU calcola:
ds:[si] - es:[di] e non **es:[di] - ds:[si]**; per evitare errori il programmatore deve necessariamente adattarsi a questa situazione in quanto la **Intel** ha imposto questo scambio di ruoli tra **SRC** e **DEST**.

Come accade per tutte le istruzioni per la manipolazione delle stringhe, anche **CMPS** e' stata concepita per operare su stringhe di grosse dimensioni; in casi del genere l'istruzione **CMPS** permette di scrivere codice molto piu' compaatto ed efficiente rispetto a quello che si ottiene con l'uso di **CMP**. Tipicamente l'istruzione **CMPS** viene utilizzata all'interno di un loop per confrontare tra loro due blocchi di memoria; un esempio pratico e' rappresentato dai programmi che eseguono la copia dei files su disco. Molti di questi programmi dispongono di un'opzione che permette di confrontare il file originale con la sua copia; in questo modo e' possibile sapere se la copia e' stata effettuata senza errori.
L'istruzione **CMPS** viene largamente utilizzata anche con le stringhe **ASCII** per effettuare confronti tra stringhe, ricerca di sottostringhe all'interno di una stringa, etc; a titolo di esempio vediamo un semplice algoritmo che ci permette di sapere se due stringhe **C** sono uguali o diverse tra loro.
L'algoritmo e' formato da un loop all'interno del quale e' presente l'istruzione **CMPS** che confronta le due stringhe carattere per carattere; la condizione di uscita dal loop e' rappresentata dal

raggiungimento della fine della stringa "sorgente" (stringhe uguali) o da una comparazione che produce **ZF=0** (stringhe diverse). Consideriamo il solito segmento dati **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** sono presenti le definizioni di due stringhe di byte chiamate **strSource** e **strDest**. Se vogliamo sapere se le due stringhe coincidono o meno possiamo scrivere il seguente algoritmo:

```
   push     ds                    ; copia ds
   pop      es                    ; in es (es ora referenzia DATASEGM)
   mov      si, offset strSource  ; ds:si punta a strSource
   mov      di, offset strDest    ; es:di punta a strDest
   cld                            ; df = 0 (incremento)
strcmp_loop:                      ; inizio del loop
   mov      al, [si]              ; per il test di fine stringa
   cmps     byte ptr [di], [si]   ; cmp byte ptr ds:[si], es:[di]
   jnz      short str_diverse     ; le stringhe sono diverse
   test     al, al                ; fine stringa ?
   jnz      short strcmp_loop     ; controllo del loop
str_uguali:                       ; strSource = strDest
```

Le prime quattro istruzioni inizializzano i registri di segmento e i registri puntatori in modo che **strSource** venga puntata da **DS:SI** e **strDest** venga puntata da **ES:DI**; l'istruzione **CLD** pone **DF=0** per la scansione in avanti delle stringhe. All'interno del loop la prima istruzione salva in **AL** l'elemento corrente di **strSource** che ci servira' per testare la condizione di fine stringa; subito dopo troviamo l'istruzione **CMPS** che compara gli elementi correnti delle due stringhe. Se si ottiene **ZF=0** i due elementi sono diversi e quindi si esce dal loop con uno short jump all'etichetta **str_diverse**; se invece **ZF=1** i due elementi sono uguali e prima di ripetere il loop bisogna verificare se e' stata raggiunta la fine di **strSource**. Visto che i due elementi sono uguali, l'eventuale fine di **strSource** implica anche la fine di **strDest**; se il test produce **ZF=0** viene ripetuto il loop, mentre in caso contrario si esce dal loop e l'elaborazione prosegue a partire dall'etichetta **str_uguali**. Questo algoritmo gestisce correttamente anche il caso di stringhe **C** di lunghezza nulla; per esercizio si puo' provare a reimplementare l'algoritmo per gestire il confronto tra due stringhe **Pascal**.

In definitiva, l'istruzione **CMPS** offre la possibilita' di effettuare comparazioni tra operandi di tipo **mem** a **8**, **16** o **32** bit; i tre codici macchina relativi a questi tre casi sono rispettivamente **A6h**, **A7h** e **66h A7h**. In relazione a queste tre possibilita' le CPU **80x86** mettono a disposizione i tre mnemonici **CMPSB**, **CMPSW** e **CMPSD**; queste tre istruzioni non richiedono nessun operando in quanto utilizzano implicitamente **DS:[SI]** come **SRC** e **ES:[DI]** come **DEST**. Come si intuisce dai mnemonici, queste tre istruzioni eseguono comparazioni tra operandi rispettivamente a **8**, **16** e **32** bit; possiamo dire quindi che:

CMPSB = CoMPare String Byte: esegue `cmp byte ptr es:[di], ds:[si]` e aggiorna **SI** e **DI** di **1** byte.
CMPSW = CoMPare String Word: esegue `cmp word ptr es:[di], ds:[si]` e aggiorna **SI** e **DI** di **2** byte.
CMPSD = CoMPare String Dword: esegue `cmp dword ptr es:[di], ds:[si]` e aggiorna **SI** e **DI** di **4** byte.

I codici macchina di queste tre istruzioni sono ovviamente gli stessi dei tre casi previsti per **CMPS**; per fare un esempio, l'istruzione:
`cmps byte ptr es:[di], ds:[si]`, puo' essere sostituita dall'istruzione:
`cmpsb`.

Quando si impiegano le istruzioni **CMPSB**, **CMPSW** e **CMPSD** la CPU utilizza automaticamente gli operandi impliciti descritti in precedenza; il programmatore non puo' alterare in alcun modo questa situazione. Se vogliamo ricorrere al segment override dobbiamo necessariamente usare l'istruzione **CMPS**; naturalmente il segment override e' consentito solo per il registro di segmento che referenzia **SRC**. Come abbiamo gia' visto in precedenza, l'esecuzione delle istruzioni **CMPS**, **CMPSB**, **CMPSW** e **CMPSD** modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF** e **CF** del **Flags Register**.

Le istruzioni INS, INSB, INSW, INSD.

Con il mnemonico **INS** si indica l'istruzione **Input from port to String** (trasferimento dati da una porta hardware ad una stringa); questa istruzione che e' disponibile solo per le CPU **80186** e superiori, richiede due operandi che svolgono il ruolo di **SRC** e **DEST**. Il programmatore deve specificare sia **SRC** che **DEST**; l'operando **SRC** deve essere l'indirizzo di una porta hardware, mentre l'operando **DEST** deve essere una locazione di memoria da **8/16/32** bit.
L'indirizzo della porta hardware deve trovarsi obbligatoriamente nel registro **DX** (**variable port**); e' proibito quindi specificare un valore immediato (**fixed port**). Ricordiamo che tutte le CPU di classe inferiore all'**80386** indirizzano le porte hardware attraverso le prime **8** linee dell'**Address Bus** (da **A0** a **A7**); con queste CPU il registro **DX** deve quindi contenere un numero di porta compreso tra **0** e **255**. Le CPU di classe **80386** e superiori invece indirizzano le porte hardware attraverso le prime **16** linee dell'**Address Bus** (da **A0** a **A15**); con queste CPU il registro **DX** deve quindi contenere un numero di porta compreso tra **0** e **65535**.
Per accedere a **DEST** la CPU utilizza obbligatoriamente la coppia **ES:DI**; e' chiaro quindi che il programmatore deve preventivamente far puntare **ES:DI** a **DEST**. In assenza di segment override la CPU utilizza ugualmente **ES** come registro di segmento predefinito; e' proibita qualsiasi altra forma di segment override.
Il codice macchina di **INS** e' formato dall'unico opcode **0110110_w**; quando la CPU incontra questo codice macchina legge **8/16/32** bit dalla porta indicata da **DX** e li trasferisce nella locazione di memoria da **1/2/4** byte puntata da **ES:DI**. Subito dopo la CPU aggiorna il puntatore **DI** in base al contenuto del flag **DF** (Direction Flag); se **DF=0** il puntatore **DI** viene incrementato di **1/2/4** byte, mentre se **DF=1** il puntatore **DI** viene decrementato di **1/2/4** byte. Per specificare la dimensione in byte del dato da leggere bisogna utilizzare il type override.

In relazione ai metodi di indirizzamento dell'operando **DEST** possiamo osservare che ci troviamo nella stessa identica situazione di **STOS**; l'operando **DEST** deve essere indirizzato obbligatoriamente attraverso **ES:DI** ed e' proibita qualsiasi altra forma di segment override.
In un precedente capitolo abbiamo conosciuto l'istruzione **IN** (input from port) che legge **8/16/32** bit da una porta hardware e li trasferisce in **AL/AX/EAX**; in termini di prestazioni l'istruzione **INS** diventa piu' conveniente di **IN** quando abbiamo la necessita' di leggere grosse quantita' di dati da una porta hardware. Generalmente i dati, man mano che vengono letti vengono trasferiti in una stringa (buffer); vediamo un esempio pratico.
Supponiamo di avere un segmento dati **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** e' presente la stringa **strBuffer** formata da **1000** byte. Se vogliamo trasferire in questa stringa **1000** byte letti dalla porta hardware numero **n** possiamo scrivere il seguente codice:

```
   push     ds                   ; copia ds
   pop      es                   ; in es (es ora referenzia DATASEGM)
   mov      di, offset strBuffer ; es:di punta a strBuffer
   mov      dx, n                ; dx = numero porta hardware
   mov      cx, 1000             ; byte da leggere
   cld                           ; df = 0 (incremento)
ins_loop:                        ; inizio del loop
   ins      byte ptr [di], dx    ; lettura di un nuovo byte
   loop     ins_loop             ; controllo del loop
```

L'istruzione **INS** offre la possibilita' di effettuare letture da **8**, **16** o **32** bit per volta; i tre codici macchina relativi a questi tre casi sono rispettivamente **6Ch**, **6Dh** e **66h 6Dh**. In relazione a queste tre possibilita' le CPU **80186** e superiori mettono a disposizione i tre mnemonici **INSB**, **INSW** e **INSD**; queste tre istruzioni non richiedono nessun operando in quanto utilizzano implicitamente **DX** come **SRC** e **ES:[DI]** come **DEST**. Come si intuisce dai mnemonici, queste tre istruzioni eseguono letture rispettivamente da **8**, **16** e **32** bit; possiamo dire quindi che:

INSB = INput from port to String Byte: esegue `ins byte ptr es:[di], dx` e aggiorna **DI** di **1** byte.
INSW = INput from port to String Word: esegue `ins word ptr es:[di], dx` e aggiorna **DI** di **2** byte.
INSD = INput from port to String Dword: esegue `ins dword ptr es:[di], dx` e aggiorna **DI** di **4** byte.

I codici macchina di queste tre istruzioni sono ovviamente gli stessi dei tre casi previsti per **INS**; per fare un esempio, l'istruzione:
`ins dword ptr es:[di], dx`, puo' essere sostituita dall'istruzione:
`insd`.
Quando si impiegano le istruzioni **INSB**, **INSW** e **INSD** la CPU utilizza automaticamente gli operandi impliciti descritti in precedenza; il programmatore non puo' alterare in alcun modo questa situazione. Lo stesso discorso vale per l'istruzione **INS** che utilizza obbligatoriamente **DX** come **SRC** e **ES:[DI]** come **DEST**; e' proibita quindi ogni altra forma di segment override. L'esecuzione delle istruzioni **INS**, **INSB**, **INSW** e **INSD** non modifica nessun campo del **Flags Register**.

Le istruzioni OUTS, OUTSB, OUTSW, OUTSD.

Con il mnemonico **OUTS** si indica l'istruzione **OUTput from String to port** (trasferimento dati da una stringa in una porta hardware); questa istruzione che e' disponibile solo per le CPU **80186** e superiori, richiede due operandi che svolgono il ruolo di **SRC** e **DEST**. Il programmatore deve specificare sia **SRC** che **DEST**; l'operando **SRC** deve essere una locazione di memoria da **8/16/32** bit, mentre l'operando **DEST** deve essere l'indirizzo di una porta hardware.
L'indirizzo della porta hardware deve trovarsi obbligatoriamente nel registro **DX** (**variable port**); e' proibito quindi specificare un valore immediato (**fixed port**).
Per accedere a **SRC** la CPU utilizza obbligatoriamente il registro **SI**; e' chiaro quindi che il programmatore deve preventivamente caricare in **SI** l'offset di **SRC**. In assenza di segment override la CPU utilizza **DS** come registro di segmento predefinito; e' consentito il segment override che permette di utilizzare in coppia con **SI** qualsiasi registro di segmento tra **CS**, **SS**, **DS**, **ES** (piu' **FS** e **GS** per le CPU **80386** e superiori).
Il codice macchina di **OUTS** e' formato dall'unico opcode **0110111_w**; quando la CPU incontra questo codice macchina legge **8/16/32** bit dalla locazione di memoria puntata da **SI** e li trasferisce nella porta hardware indicata da **DX**. Subito dopo la CPU aggiorna il puntatore **SI** in base al contenuto del flag **DF** (Direction Flag); se **DF=0** il puntatore **SI** viene incrementato di **1/2/4** byte, mentre se **DF=1** il puntatore **SI** viene decrementato di **1/2/4** byte. Per specificare la dimensione in byte del dato da scrivere bisogna utilizzare il type override.

In relazione ai metodi di indirizzamento dell'operando **SRC** possiamo osservare che ci troviamo nella stessa identica situazione di **LODS**; l'operando **SRC** deve essere indirizzato obbligatoriamente attraverso **SI** (con registro di segmento predefinito **DS**) ed e' consentito il segment override.
In un precedente capitolo abbiamo conosciuto l'istruzione **OUT** (output to port) che legge **8/16/32** bit da **AL/AX/EAX** e li trasferisce in una porta hardware; in termini di prestazioni l'istruzione **OUTS** diventa piu' conveniente di **OUT** quando abbiamo la necessita' di scrivere grosse quantita' di

dati in una porta hardware. Generalmente i dati da inviare alla porta hardware vengono letti da una stringa (buffer); vediamo un esempio pratico.
Supponiamo di avere un segmento dati **DATASEGM** referenziato da **DS**; all'interno di **DATASEGM** e' presente la stringa **strBuffer** formata da **1000** byte. Se vogliamo inviare questi **1000** byte alla porta hardware numero **n** possiamo scrivere il seguente codice:

```
   mov      si, offset strBuffer ; ds:si punta a strBuffer
   mov      dx, n                ; dx = numero porta hardware
   mov      cx, 1000             ; byte da scrivere
   cld                           ; df = 0 (incremento)
outs_loop:                       ; inizio del loop
   outs     dx, byte ptr [si]    ; scrittura di un nuovo byte
   loop     outs_loop            ; controllo del loop
```

L'istruzione **OUTS** offre la possibilita' di effettuare scritture da **8**, **16** o **32** bit per volta; i tre codici macchina relativi a questi tre casi sono rispettivamente **6Eh**, **6Fh** e **66h 6Fh**. In relazione a queste tre possibilita' le CPU **80186** e superiori mettono a disposizione i tre mnemonici **OUTSB**, **OUTSW** e **OUTSD**; queste tre istruzioni non richiedono nessun operando in quanto utilizzano implicitamente **[SI]** come **SRC** e **DX** come **DEST**. Come si intuisce dai mnemonici, queste tre istruzioni eseguono scritture rispettivamente da **8**, **16** e **32** bit; possiamo dire quindi che:

OUTSB = OUTput from String Byte to port: esegue `outs dx, byte ptr [si]` e aggiorna **SI** di **1** byte.
OUTSW = OUTput from String Word to port: esegue `outs dx, word ptr [si]` e aggiorna **SI** di **2** byte.
OUTSD = OUTput from String Dword to port: esegue `outs dx, dword ptr [si]` e aggiorna **SI** di **4** byte.

I codici macchina di queste tre istruzioni sono ovviamente gli stessi dei tre casi previsti per **OUTS**; per fare un esempio, l'istruzione:
`outs dx, byte ptr [si]`, puo' essere sostituita dall'istruzione:
`outsb`.
Quando si impiegano le istruzioni **OUTSB**, **OUTSW** e **OUTSD** la CPU utilizza automaticamente gli operandi impliciti descritti in precedenza; il programmatore non puo' alterare in alcun modo questa situazione. Se vogliamo ricorrere al segment override dobbiamo necessariamente usare l'istruzione **OUTS**; naturalmente il segment override e' consentito solo per il registro di segmento che referenzia **SRC**. L'esecuzione delle istruzioni **OUTS**, **OUTSB**, **OUTSW** e **OUTSD** non modifica nessun campo del **Flags Register**.

I prefissi REP, REPE/REPZ, REPNE/REPNZ.

In base a tutte le considerazioni svolte in questo capitolo, si capisce subito che le istruzioni per le stringhe sono state concepite espressamente per operare su stringhe di dimensioni medio grandi; nel caso invece di stringhe di piccole dimensioni, queste istruzioni devono essere sostituite con le istruzioni equivalenti illustrate nei precedenti capitoli, che permettono di scrivere codice molto piu' compatto e veloce. Un'altro aspetto che e' emerso in modo chiaro e' dato dal fatto che per operare efficacemente su grossi blocchi di dati, le istruzioni per le stringhe vengono usualmente inserite all'interno di un loop; si tratta di una situazione talmente frequente da indurre la **Intel** a creare appositi prefissi che applicati a queste istruzioni permettono di automatizzare alcune operazioni fondamentali per il controllo dei loop. I tre prefissi disponibili vengono indicati con i tre mnemonici **REP**, **REPE/REPZ** e **REPNE/REPNZ**; con questi mnemonici viene indicata l'istruzione **REPeat following string operation** (iterazione dell'istruzione per le stringhe che segue il prefisso). Il codice

macchina generale di questi prefissi e' formato dal solo opcode **1111001_z** seguito poi dall'opcode dell'istruzione per le stringhe; il bit indicato con **z** viene chiamato **zero bit**. Se **z=1** la CPU deve verificare la condizione **ZF=1** per ripetere il loop, mentre Se **z=0** la CPU deve verificare la condizione **ZF=0** per ripetere il loop; naturalmente queste considerazioni si applicano solo alle istruzioni **SCAS** e **CMPS**.
Quando la CPU incontra il codice macchina di un prefisso associato ad una operazione per le stringhe, attiva un loop nel quale l'istruzione per le stringhe, oltre a svolgere la propria funzione, gestisce anche l'aggiornamento dei puntatori, mentre il prefisso gestisce la condizione di uscita dal loop legata al contenuto di **CX** e in alcuni casi al contenuto di **ZF**; in sostanza possiamo dire che gli effetti prodotti dai tre prefissi **REP**, **REPE/REPZ** e **REPNE/REPNZ** sono molto simili a quelli prodotti dalle istruzioni **LOOP**, **LOOPE/LOOPZ** e **LOOPNE/LOOPNZ**.

Analizziamo in dettaglio il lavoro che svolgono questi prefissi quando vengono applicati alle istruzioni per le stringhe; a partire dalle CPU **80386** e superiori i prefissi eseguono in sequenza le seguenti operazioni:

1) Controllo del contatore **CX**. Se **CX=0** il loop termina e l'esecuzione salta al punto **7**; in sostanza questi prefissi eseguono automaticamente una sorta di **JCXZ**.
2) Elaborazione delle **maskable interrupts**. Tutte le interruzioni mascherabili in attesa vengono elaborate dalla CPU e vengono poi sospese sino alla prossima iterazione (o all'uscita dal loop).
3) Esecuzione singola dell'istruzione per le stringhe.
4) Decremento contatore. **CX** viene decrementato di **1** senza produrre nessun effetto sul **Flags Register**.
5) Controllo del flag **ZF** (solo per **SCAS** e **CMPS**). Se il prefisso e' **REPE/REPZ** e **ZF=0** il loop viene terminato e l'esecuzione salta al punto **7**; se il prefisso e' **REPNE/REPNZ** e **ZF=1** il loop viene terminato e l'esecuzione salta al punto **7**.
6) Controllo del loop. L'esecuzione riprende dal punto **1**.
7) Istruzione successiva al loop.

Come e' stato appena detto, questa sequenza di operazioni si riferisce alla gestione dei prefissi **REP**, **REPE/REPZ**, **REPNE/REPNZ** da parte delle CPU **80386** e superiori; le CPU di classe inferiore non eseguono il passo **2** (elaborazione sincronizzata delle interruzioni mascherabili). I progettisti delle CPU precedenti l'**80386** si sono accorti troppo tardi di un grave problema che si verifica quando sopraggiunge una richiesta di interruzione hardware nel bel mezzo di un loop gestito da un prefisso **REP**, **REPE/REPZ**, **REPNE/REPNZ** associato ad una istruzione per le stringhe che fa uso del segment override; in un caso del genere, al termine della **ISR** che elabora l'interruzione, la CPU riprende il loop dimenticandosi pero' del segment override e indirizzando quindi le stringhe con i registri di segmento predefiniti. La conseguenza di tutto cio' e' che al termine del loop si ottengono risultati totalmente privi di senso; in base a queste considerazioni, quando si utilizza una CPU **80286** o inferiore bisogna evitare nella maniera piu' assoluta di incorrere nella situazione appena descritta. La soluzione al problema consiste semplicemente nel servirsi di un loop ordinario gestito da istruzioni come **LOOP**, **Jcond**, **JMP**, etc; in assenza di segment override invece, l'uso dei prefissi e' assolutamente affidabile.

Vediamo ora alcuni esempi che illustrano in pratica il principio di funzionamento dei prefissi **REP**, **REPE/REPZ**, **REPNE/REPNZ**; cominciamo con **REP** che rappresenta il caso piu' semplice in quanto questo prefisso si limita ad attivare un loop controllato da **CX**. Teoricamente il prefisso **REP** puo' essere applicato a tutte le istruzioni per le stringhe anche se non ha molto senso applicarlo a **SCAS** e **CMPS**; per questi due casi diventa importante lo stato del flag **ZF** per cui bisogna utilizzare **REPE/REPZ** o **REPNE/REPNZ**. L'uso dei prefissi non ha molto senso nemmeno per l'istruzione **LODS** che, come abbiamo visto in precedenza, legge dati da una stringa e li carica

nell'accumulatore; generalmente, questi dati appena letti vengono sottoposti a verifiche e successive elaborazioni. Per poter svolgere questo lavoro e' necessario inserire l'istruzione **LODS** all'interno di un loop classico, gestito attraverso istruzioni come **LOOP**, **Jcond** o **JMP**; in definitiva possiamo dire che il prefisso **REP** e' particolarmente indicato per le istruzioni **MOVS**, **STOS**, **INS** e **OUTS**.
Vediamo un esempio relativo a **MOVS** che ci permette di effettuare una copia di una stringa **Pascal**; supponiamo di avere un blocco **DATASEGM** referenziato da **DS**. All'interno di **DATASEGM** sono presenti le definizioni di due stringhe **Pascal** chiamate **strSource** e **strDest**, con **strDest** che ha spazio sufficiente per contenere **strSource**; ricordando che **strSource[0]** contiene la lunghezza della stringa possiamo scrivere:

```
   push     ds                   ; copia ds
   pop      es                   ; in es (es ora referenzia DATASEGM)
   mov      si, offset strSource ; ds:si punta a strSource
   mov      di, offset strDest   ; es:di punta a strDest
   mov      cx, [si]             ; cx = lunghezza in byte di strSource
   cld                           ; df = 0 (incremento)
   rep      movsb                ; ripete movsb cx volte
```

Come si puo' notare si ottiene un codice estremamente compatto che diventa anche molto efficiente nel caso di stringhe di grosse dimensioni; il prefisso **REP** si occupa del controllo del loop attraverso **CX**, mentre **MOVSB** si occupa tra l'altro dell'aggiornamento dei puntatori **SI** e **DI**. Questa tecnica diventa potentissima nel caso di grossi blocchi di dati; ricordando che in modalita' reale a **16** bit gli offset possono andare da **0000h** a **FFFFh**, possiamo dire che usando **REP** in combinazione con **MOVS** siamo in grado di spostare da una parte all'altra della memoria sino a **64** Kb di dati in un colpo solo!

Se **strSource** e **strDest** fossero state due stringhe **C** avremmo dovuto scrivere:

```
   push     ds                   ; copia ds
   pop      es                   ; in es (es ora referenzia DATASEGM)
   mov      si, offset strSource ; ds:si punta a strSource
   mov      di, offset strDest   ; es:di punta a strDest
   cld                           ; df = 0 (incremento)
strcopy_loop:                    ; inizio del loop
   mov      al, [si]             ; per il test di fine stringa
   movsb                         ; movs byte ptr es:[di], ds:[si]
   test     al, al               ; al = 0 ?
   jnz      strcpy_loop          ; controllo del loop
```

La necessita' di effettuare il test di fine stringa ci impedisce di utilizzare il prefisso **REP**; questa e' un'ulteriore dimostrazione della maggiore semplicita' di gestione delle stringhe **Pascal** rispetto alle stringhe **C**.

Un'altra istruzione per le stringhe particolarmente adatta all'uso combinato con **REP** e' **STOS**; come abbiamo visto in un precedente esempio, **STOS** viene largamente utilizzata per inizializzare o reinizializzare grossi blocchi di memoria. Supponiamo ad esempio di avere una stringa da **10000** dword (**40000** byte) chiamata **strBuffer** e definita in un blocco **DATASEGM** referenziato da **ES**; se vogliamo inizializzare tutti gli elementi di **strBuffer** con il valore **0** possiamo scrivere:

```
   mov      di, offset strBuffer ; es:di punta a strBuffer
   xor      eax, eax             ; operando sorgente eax = 0
   mov      cx, 10000            ; cx = 10000 dword
   cld                           ; df = 0 (incremento)
   rep      stosd                ; ripete stosd cx volte
```

Passiamo ora ai prefissi **REPE/REPZ** e **REPNE/REPNZ**; come e' stato gia' detto questi prefissi sono stati concepiti espressamente per le istruzioni **SCAS** e **CMPS**. Il prefisso **REPE/REPZ** attiva un loop che viene ripetuto purche' **CX** sia diverso da **0** e **ZF=1**; analogamente, il prefisso **REPNE/REPNZ** attiva un loop che viene ripetuto purche' **CX** sia diverso da **0** e **ZF=0**. Vediamo un'applicazione pratica della coppia **REPNE/REPNZ** e **SCAS** per il calcolo della lunghezza di una stringa **C**; naturalmente nel caso di stringhe **Pascal** questo calcolo e' immediato. Se la stringa **C** si chiama **strSource** ed e' definita in un blocco **DATASEGM** referenziato da **DS** possiamo scrivere:

```
   push     ds                        ; copia ds
   pop      es                        ; in es (es ora referenzia DATASEGM)
   mov      di, offset strSource      ; es:di punta a strSource
   xor      al, al                    ; al = 0 (da comparare con con es:[di])
   mov      cx, 0FFFFh                ; massima lunghezza possibile
   cld                                ; df = 0 (incremento)
   repnz    scasb                     ; ripete scasb se ZF=0 e CX diverso da 0
   dec      di                        ; elimina l'incremento finale
   sub      di, offset strSource      ; di = lunghezza stringa C
```

Per concludere vediamo un esempio pratico relativo alla coppia **REPE/REPZ** e **CMPS** utilizzata per confrontare tra loro due stringhe **Pascal**; supponiamo di avere il solito blocco **DATASEGM** referenziato da **DS**. All'interno di **DATASEGM** sono presenti due stringhe **Pascal** chiamate **strSource** e **strDest**; per sapere se queste due stringhe sono uguali o meno possiamo scrivere:

```
   push     ds                     ; copia ds
   pop      es                     ; in es (es ora referenzia DATASEGM)
   mov      si, offset strSource ; ds:si punta a strSource
   mov      di, offset strDest   ; es:di punta a strDest
   mov      cx, [si]               ; cx = lunghezza strSource
   cld                             ; df = 0 (incremento)
   repe     cmpsb                  ; ripete cmpsb se ZF=1 e CX diverso da 0
   je       short str_uguali       ; se ZF = 1 le due stringhe sono uguali
str_diverse:                       ; altrimenti sono differenti
```

Il loop termina se **CX=0** e/o se **ZF=0**; se il loop termina prematuramente perche' **ZF=0**, le due stringhe sono differenti. Se il loop termina perche' **CX=0**, dobbiamo controllare lo stato finale di **ZF**; se **ZF=0** le due stringhe sono differenti, mentre se **ZF=1** le due stringhe sono uguali.
Osserviamo che il primo confronto coinvolge **strSource[0]** e **strDest[0]** che contengono le lunghezze in byte delle due stringhe; se si ottiene **ZF=0** le due stringhe hanno lunghezze differenti e sono quindi diverse tra loro.
Questa stessa tecnica e' applicabile anche al confronto tra due stringhe **C** purche' si conosca in anticipo la loro lunghezza; se non conosciamo questa informazione dobbiamo ricorrere all'esempio mostrato in questo capitolo per l'istruzione **CMPS**.

Capitolo 23 - Istruzioni varie

In questo capitolo viene brevemente descritto il principio di funzionamento di varie istruzioni della CPU che in genere compaiono meno frequentemente nei programmi Assembly; per maggiori dettagli si consiglia di consultare i manuali dei vari assembler o la documentazione tecnica relativa ai vari modelli di CPU.

L'istruzione BOUND.

Con il mnemonico **BOUND** si indica l'istruzione **check array index against BOUNDs** (verifica dei limiti dell'indice di un vettore); questa istruzione, che e' disponibile solo per le CPU **80186** e superiori, richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **DEST** deve essere un registro contenente l'indice da verificare; l'operando **SRC** deve essere una locazione di memoria contenente il limite inferiore e il limite superiore dell'indice del vettore. Il codice macchina dell'istruzione **BOUND** e' formato dai due opcodes **01100010b = 62h** e **mod_reg_r/m**; quando la CPU incontra questo codice macchina, verifica che il contenuto di **DEST** rientri nei limiti specificati da **SRC**. In caso affermativo, l'esecuzione del programma prosegue con l'istruzione successiva a **BOUND**; in caso negativo (indice fuori limite) viene chiamata l'**INT 05h**. In genere, l'**ISR** predefinita associata all'**INT 05h** contiene una semplice istruzione **IRET**; il programmatore puo' quindi installare una propria **ISR** che gestisce la condizione di indice fuori limite di un vettore. Per effettuare questo tipo di verifica la CPU tratta l'indice di un vettore come un numero con segno; in questo modo e' possibile gestire anche il caso di un indice che "sconfina" ad esempio al di sotto dello zero.
E' importante tener presente che in relazione all'istruzione **BOUND**, il termine **indice** rappresenta non la posizione ma bensi' l'offset di un elemento di un vettore; il programmatore deve quindi specificare i limiti inferiore e superiore dell'indice, in funzione del metodo di indirizzamento utilizzato. Supponiamo ad esempio di avere un vettore **vectWord** formato da **10** word e definito all'offset **0020h** del blocco dati del nostro programma; i **10** elementi che formano **vectWord** si vengono a trovare quindi nell'ordine agli offset **0020h**, **0022h**, **0024h** e cosi' via, sino all'ultimo elemento che si trova all'offset **0032h**. Se vogliamo accedere a **vectWord** attraverso il registro puntatore **BX** possiamo scrivere:
`mov bx, offset vectWord`; a questo punto **BX** contiene l'offset **0020h** del primo elemento di **vectWord**. Per muoverci all'interno di **vectWord** dobbiamo assegnare a **BX** tutti i valori compresi tra **0020h** (primo elemento) e **0032h** (ultimo elemento); possiamo dire quindi che in questo caso, il limite inferiore di **BX** e' **0020h**, mentre il limite superiore e' **0032h**. Attraverso l'istruzione **BOUND** possiamo sapere se in fase di esecuzione del nostro porgramma (**run time**) si verifica uno sconfinamento del registro **BX**; prima di tutto dobbiamo definire un'apposita variabile contenente i limiti inferiore e superiore; possiamo scrivere ad esempio:
`boundData dd 00320020h`. Come si puo' notare, **boundData** e' una variabile a **32** bit che contiene il limite inferiore nella word meno significativa, e il limite superiore nella word piu' significativa; a questo punto all'interno di un loop possiamo scrivere ad esempio:

```
init_loop:
   bound    bx, boundData     ; verifica dei limiti
   mov      [bx], ax          ; trasferimento dati
   add      bx, 2             ; indice prossimo elemento
   loop     init_loop         ; controllo del loop
```

Come si puo' notare da questo esempio, la cosa migliore da fare consiste nell'inserire l'istruzione **BOUND** immediatamente prima dell'istruzione che accede agli elementi del vettore; in questo modo si scongiura l'eventualita' di scrivere dati in aree di memoria che non appartengono al vettore.

Se il contenuto di **BX** e' compreso tra **0020h** e **0032h**, l'esecuzione prosegue normalmente; se invece il contenuto di **BX** e' minore di **0020h** o maggiore di **0032h**, viene chiamata l'**INT 05h**. Volendo accedere agli elementi di **vectWord** attraverso la notazione **vectWord[bx]**, dobbiamo inizializzare **BX** con il valore **0**; per poter "esplorare" tutto il vettore, dobbiamo assegnare a **BX** i valori **0000h**, **0002h**, **0004h** e cosi' via, sino a **0012h** che e' l'indice dell'ultimo elemento. In questo caso i limiti dell'indice **BX** sono rappresentati dal valore **0000h** (limite inferiore) e dal valore **0012h** (limite superiore); la variabile **boundData** deve essere quindi ridefinita come: `boundData dd 00120000h`. Come e' stato detto in precedenza, la CPU tratta l'indice come un numero con segno in modo da gestire anche il caso di sconfinamento al di sotto del limite inferiore; se abbiamo ad esempio **BX=0000h**, sottraendo **2** byte a **BX** otteniamo **BX=FFFEh** che per i numeri con segno a **16** bit e' la rappresentazione in complemento a due di **-2**. In questo caso la CPU eseguendo l'istruzione **BOUND** e' in grado di rilevare che siamo andati sotto il limite inferiore **0**.

Nella gestione di un vettore, l'indice fuori limite non sempre rappresenta una situazione di errore; puo' capitare ad esempio di avere la necessita' di sconfinare consapevolmente dai limiti di un vettore. Questa situazione e' del tutto normale per i programmatori **C** e **Assembly**; d'altra parte la filosofia di questi linguaggi consiste proprio nel fidarsi del programmatore in modo da consentirgli la massima liberta' d'azione. Se invece l'indice di un vettore va fuori limite all'insaputa del programmatore, ci troviamo davanti ad una grave situazione di errore; se c'e' il sospetto che possa verificarsi un problema di questo genere e' opportuno utilizzare l'istruzione **BOUND**. In tal caso il programmatore deve anche ricordarsi di installare l'apposita **ISR** attraverso la quale e' possibile prendere le decisioni piu' opportune; generalmente il lavoro svolto dalla **ISR** consiste nel terminare prematuramente il programma dopo aver mostrato sullo schermo un messaggio di errore.
Se decidiamo di trattare l'indice fuori limite come una situazione di errore, dobbiamo prestare molta attenzione al particolare metodo che la CPU utilizza per eseguire l'istruzione **BOUND**; vediamo infatti quello che succede quando la CPU si accorge che l'indice del vettore e' fuori limite. Prima di chiamare l'**INT 05h** la CPU salva come al solito nello stack il contenuto del registro **FLAGS** e il contenuto corrente del registro **CS**; a questo punto la CPU invece di salvare nello stack l'offset dell'istruzione successiva a **BOUND**, salva l'offset dell'istruzione **BOUND**. Questo significa che al termine dell'**ISR**, l'esecuzione riprende dall'indirizzo della stessa istruzione **BOUND** che ha determinato la chiamata dell'**INT 05h**; se il programmatore non prende provvedimenti, viene rieseguita l'istruzione **BOUND** che determina una nuova chiamata dell'**INT 05h**. In sostanza, l'**INT 05h** viene chiamata a ripetizione e il nostro programma entra in un loop infinito determinando il blocco del computer; proprio per questo motivo e' opportuno che l'**ISR** installata dal programmatore proceda alla terminazione prematura del nostro programma.

Con le CPU a **16** bit l'istruzione **BOUND** richiede un registro a **16** bit come **DEST** e una locazione di memoria a **32** bit come **SRC**; nel caso delle CPU **80386** e superiori e' anche possibile utilizzare un registro a **32** bit come **DEST** e una locazione di memoria a **64** bit come **SRC**. L'esecuzione dell'istruzione **BOUND** non altera nessun campo del **Flags Register**.

L'istruzione BSF.

Con il mnemonico **BSF** si indica l'istruzione **Bit Scan Forward** (ricerca in avanti del primo bit di **DEST**) che vale **1**); questa istruzione, che e' disponibile solo per le CPU **80386** e superiori, richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **SRC** deve essere di tipo **reg/mem**, mentre l'operando **DEST** deve essere di tipo **reg**. Il codice macchina dell'istruzione **BSF** e' formato dagli opcodes **00001111b**, **10111100b** e **mod_reg_r/m**; quando la CPU incontra questo codice macchina, esegue una scansione in avanti di **SRC** alla ricerca del primo bit che vale **1**; la scansione inizia quindi dal bit meno significativo di **SRC**. Se non viene trovato nessun bit che vale **1** (**SRC=0**), la CPU pone **ZF=0**; se invece, alla posizione **n** di **SRC** viene incontrato un bit che

vale **1**, la CPU pone **ZF=1** e salva il valore **n** in **DEST**.
Supponiamo ad esempio di avere la seguente definizione:
`varWord dw 0011001100100000b`; come si puo' notare, partendo dal bit meno significativo (posizione **0**), alla posizione **5** di **varWord** si incontra il primo bit che vale **1**. Quando la CPU incontra il codice macchina dell'istruzione:
`bsf cx, varWord`, pone quindi **ZF=1** e carica in **CX** il valore **5** che rappresenta appunto la posizione del primo bit di **varWord** che vale **1**.
L'istruzione **BSF** lavora anche con operandi a **32** bit; l'esecuzione dell'istruzione **BSF** altera solo il campo **ZF** del **Flags Register**.

L'istruzione BSR.

Con il mnemonico **BSR** si indica l'istruzione **Bit Scan Reverse** (ricerca all'indietro del primo bit di **DEST** che vale **1**); questa istruzione, che e' disponibile solo per le CPU **80386** e superiori, richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **SRC** deve essere di tipo **reg/mem**, mentre l'operando **DEST** deve essere di tipo **reg**. Il codice macchina dell'istruzione **BSF** e' formato dagli opcodes **00001111b**, **10111101b** e **mod_reg_r/m**; quando la CPU incontra questo codice macchina, esegue una scansione all'indietro di **SRC** alla ricerca del primo bit che vale **1**; la scansione inizia quindi dal bit piu' significativo di **SRC**. Se non viene trovato nessun bit che vale **1** (**SRC=0**), la CPU pone **ZF=0**; se invece, alla posizione **n** di **SRC** viene incontrato un bit che vale **1**, la CPU pone **ZF=1** e salva il valore **n** in **DEST**.
Supponiamo ad esempio di avere la seguente definizione:
`varWord dw 0011001100100000b`; come si puo' notare, partendo dal bit piu' significativo (posizione **15**), alla posizione **13** di **varWord** si incontra il primo bit che vale **1**. Quando la CPU incontra il codice macchina dell'istruzione:
`bsr cx, varWord`, pone quindi **ZF=1** e carica in **CX** il valore **13** che rappresenta appunto la posizione del primo bit di **varWord** che vale **1**.
L'istruzione **BSR** lavora anche con operandi a **32** bit; l'esecuzione dell'istruzione **BSR** altera solo il campo **ZF** del **Flags Register**.

L'istruzione BSWAP.

Con il mnemonico **BSWAP** si indica l'istruzione **Byte SWAP** (scambio di byte); questa istruzione, che e' disponibile solo per le CPU **80486** e superiori, richiede esplicitamente un unico operando che deve essere un registro a **32** bit. Il codice macchina di **BSWAP** e' formato dai due opcodes **00001111b** e **11001_reg**; quando la CPU incontra questo codice macchina, accede al contenuto dell'operando **reg32**, legge i **4** byte che occupano le posizioni **0**, **1**, **2** e **3** e li reinserisce in **reg32** nelle posizioni **3**, **2**, **1** e **0**.
Supponiamo ad esempio di avere **EDX = 03F1A23Fh**; subito dopo l'esecuzione dell'istruzione:
`bswap edx`, otteniamo **EDX=3FA2F103h**.
L'istruzione **BSWAP** puo' essere impiegata quindi per conversioni da **LITTLE-ENDIAN** a **BIG-ENDIAN** e viceversa; se si utilizza **BSWAP** con un operando di ampiezza inferiore a **32** bit si ottengono risultati privi di senso. Se vogliamo scambiare i due byte di un registro a **16** bit dobbiamo utilizzare **XCHG**; nela caso ad esempio del registro **DX** possiamo scrivere:
`xchg dh, dl`.
Per poter utilizzare l'istruzione **BSWAP** dobbiamo ovviamente inserire la direttiva `.486`; se stiamo utilizzando un vecchio assembler che non supporta questa direttiva, dobbiamo inserire direttamente il codice macchina dell'istruzione. Nel caso dell'esempio precedente, tenendo conto dell'operand size prefix **01010101b = 66h** e del fatto che **EDX = 010b**, dobbiamo scrivere:
`db 01010101b, 00001111b, 11001010b ; bswap edx`.
L'esecuzione dell'istruzione **BSWAP** non altera nessun campo del **Flags Register**.

L'istruzione <u>BT</u>.

Con il mnemonico **BT** si indica l'istruzione **Bit Test** (test di un bit); questa istruzione che e' disponibile solo per le CPU **80386** e superiori richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **DEST** deve essere di tipo **reg/mem**, mentre l'operando **SRC** puo' essere di tipo **reg** o **imm8**. Se **SRC** e' di tipo **reg** il codice macchina di **BT** e' formato dagli opcodes **00001111b**, **10100011**, **mod_reg_r/m**; se **SRC** e' di tipo **imm8** il codice macchina di **BT** e' formato dagli opcodes **00001111b**, **10111010**, **mod_100_r/m** seguiti da **imm8**. Quando la CPU incontra questo codice macchina legge il bit di **DEST** che si trova in posizione **SRC** e lo copia in **CF** (Carry Flag).
Supponiamo ad esempio di avere **CX = 0011110011110000b** e **DX = 6**; eseguendo l'istruzione:
`bt cx, dx` otteniamo **CF = 1**. Osserviamo infatti che il bit che in **CX** occupa la posizione **6** vale **1**; in sostanza il valore contenuto in **SRC** rappresenta un offset in bit all'interno di **DEST**.
L'esecuzione di **BT** modifica solo il campo **CF** del **Flags Register**.

L'istruzione <u>BTC</u>.

Con il mnemonico **BTC** si indica l'istruzione **Bit Test and Complement** (test e inversione di un bit); questa istruzione che e' disponibile solo per le CPU **80386** e superiori richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **DEST** deve essere di tipo **reg/mem**, mentre l'operando **SRC** puo' essere di tipo **reg** o **imm8**. Se **SRC** e' di tipo **reg** il codice macchina di **BTC** e' formato dagli opcodes **00001111b**, **10111011**, **mod_reg_r/m**; se **SRC** e' di tipo **imm8** il codice macchina di **BTC** e' formato dagli opcodes **00001111b**, **10111010**, **mod_111_r/m** seguiti da **imm8**. Quando la CPU incontra questo codice macchina legge il bit di **DEST** che si trova in posizione **SRC**, lo copia in **CF** (Carry Flag) e poi complementa lo stesso bit di **DEST**.
Supponiamo ad esempio di avere **CX = 0011110011110000b** e **DX = 6**; eseguendo l'istruzione:
`btc cx, dx` otteniamo **CF = 1** e **CX = 0011110010110000b**. Osserviamo infatti che il bit che in **CX** occupa la posizione **6** vale **1**; dopo aver copiato questo bit in **CF** la CPU lo complementa per cui il bit di **CX** in posizione **6** diventa **0**. L'esecuzione di **BTC** modifica solo il campo **CF** del **Flags Register**.

L'istruzione <u>BTR</u>.

Con il mnemonico **BTR** si indica l'istruzione **Bit Test and Reset** (test e azzeramento di un bit); questa istruzione che e' disponibile solo per le CPU **80386** e superiori richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **DEST** deve essere di tipo **reg/mem**, mentre l'operando **SRC** puo' essere di tipo **reg** o **imm8**. Se **SRC** e' di tipo **reg** il codice macchina di **BTR** e' formato dagli opcodes **00001111b**, **10110011**, **mod_reg_r/m**; se **SRC** e' di tipo **imm8** il codice macchina di **BTR** e' formato dagli opcodes **00001111b**, **10111010**, **mod_110_r/m** seguiti da **imm8**. Quando la CPU incontra questo codice macchina legge il bit di **DEST** che si trova in posizione **SRC**, lo copia in **CF** (Carry Flag) e poi azzera lo stesso bit di **DEST**.
Supponiamo ad esempio di avere **CX = 0011110011110000b** e **DX = 6**; eseguendo l'istruzione:
`btr cx, dx` otteniamo **CF = 1** e **CX = 0011110010110000b**. Osserviamo infatti che il bit che in **CX** occupa la posizione **6** vale **1**; dopo aver copiato questo bit in **CF** la CPU lo azzera per cui il bit di **CX** in posizione **6** diventa **0**. L'esecuzione di **BTR** modifica solo il campo **CF** del **Flags Register**.

L'istruzione <u>BTS</u>.

Con il mnemonico **BTS** si indica l'istruzione **Bit Test and Set** (test e attivazione di un bit); questa

istruzione che e' disponibile solo per le CPU **80386** e superiori richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **DEST** deve essere di tipo **reg/mem**, mentre l'operando **SRC** puo' essere di tipo **reg** o **imm8**. Se **SRC** e' di tipo **reg** il codice macchina di **BTS** e' formato dagli opcodes **00001111b**, **10111011**, **mod_reg_r/m**; se **SRC** e' di tipo **imm8** il codice macchina di **BTS** e' formato dagli opcodes **00001111b**, **10111010**, **mod_101_r/m** seguiti da **imm8**. Quando la CPU incontra questo codice macchina legge il bit di **DEST** che si trova in posizione **SRC**, lo copia in **CF** (Carry Flag) e poi pone a **1** lo stesso bit di **DEST**.
Supponiamo ad esempio di avere **CX = 0011110011110000b** e **DX = 2**; eseguendo l'istruzione: `bts cx, dx` otteniamo **CF = 0** e **CX = 0011110011110100b**. Osserviamo infatti che il bit che in **CX** occupa la posizione **2** vale **0**; dopo aver copiato questo bit in **CF** la CPU lo pone a **1** per cui il bit di **CX** in posizione **2** diventa **1**. L'esecuzione di **BTS** modifica solo il campo **CF** del **Flags Register**.

L'istruzione CMPXCHG.

Con il mnemonico **CMPXCHG** si indica l'istruzione **CoMPare and eXCHanGe** (compara e scambia); questa istruzione che e' disponibile solo per le CPU **80486** e superiori richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **DEST** deve essere di tipo **reg/mem**, mentre l'operando **SRC** deve essere di tipo **reg**; il codice macchina di **CMPXCHG** e' formato dagli opcodes **00001111b**, **1011000_w**, **mod_reg_r/m**. Quando la CPU incontra questo codice macchina compara **AL/AX/EAX** con **DEST**; se il risultato e' zero la CPU pone **ZF=1** e carica **SRC** in **DEST**. Se il risultato e' diverso da zero la CPU pone **ZF=0** e carica **DEST** in **AL/AX/EAX**; in base all'ampiezza in bit degli operandi la CPU decide se utilizzare **AL**, **AX** o **EAX**.
Vediamo un esempio pratico:

```
.486                  ; set di istruzioni dell'80486
mov      ax, 3FD0h    ; accumulatore
mov      bx, 2DA1h    ; operando destinazione
mov      cx, 3FF0h    ; operando sorgente
cmpxchg  bx, cx       ; esegue ax - bx
```

L'istruzione **CMPXCHG** confronta **AX** con **BX**, e siccome **AX** e' diverso da **BX**, la CPU pone **ZF=0** e carica **BX** in **AX**; alla fine si ottiene quindi **AX=2DA1h**, **BX=2DA1h**, **CX=3FF0h** e **ZF=0**. Se **AX** fosse stato uguale a **BX** (**2DA1h**) la CPU avrebbe posto **ZF=1** e avrebbe caricato **CX** in **BX**; alla fine avremmo ottenuto **AX=2DA1h**, **BX=3FF0h**, **CX=3FF0h** e **ZF=1**.
Analogamente a quanto accade con **CMP**, anche l'esecuzione di **CMPXCHG** modifica i campi **OF**, **SF**, **ZF**, **AF**, **PF**, **CF** del **Flags Register**.

L'istruzione CMPXCHG8B.

Con il mnemonico **CMPXCHG8B** si indica l'istruzione **CoMPare and eXCHanGe 8 Byte** (compara e scambia due operandi a **64** bit); questa istruzione che e' disponibile solo per le CPU **80586** e superiori richiede esplicitamente un operando che svolge il ruolo di **DEST**. L'operando **DEST** deve essere esclusivamente di tipo **reg64/mem64** dove **r64** indica un registro a **64** bit della **FPU**; come operando **SRC** la CPU utilizza implicitamente la coppia **ECX:EBX** con **EBX** che rappresenta la dword meno significativa, e **ECX** la dword piu' significativa. Per la comparazione la CPU utilizza implicitamente la coppia **EDX:EAX** con **EAX** che rappresenta la dword meno significativa, e **EDX** la dword piu' significativa. Il codice macchina di **CMPXCHG8B** e' formato dagli opcodes **00001111b**, **11000111**, **mod_reg_r/m**; quando la CPU incontra questo codice macchina compara **EDX:EAX** con **DEST**. Se il risultato e' zero la CPU pone **ZF=1** e carica in **DEST** il contenuto di **ECX:EBX**; se il risultato e' diverso da zero la CPU pone **ZF=0** e carica in

EDX:EAX il contenuto di **DEST**.
Vediamo un esempio pratico; prima di tutto definiamo la seguente variabile a **64** bit:
`varQword dq 3F2D40312883F890h;`
a questo punto possiamo scrivere:

```
.586                          ; set di istruzioni dell'80586
mov         ecx, 22FF4481h    ; dword alta di SRC
mov         ebx, 3DF1AAB0h    ; dword bassa di SRC
mov         edx, 3F2D4031h    ; dword alta del sottraendo
mov         eax, 2883F890h    ; dword bassa del sottraendo
cmpxchg8b   varQword          ; esegue edx:eax - varQword
```

L'istruzione **CMPXCHG8B** confronta **EDX:EAX** con **varQword**, e siccome **EDX:EAX** e' uguale a **varQword**, la CPU pone **ZF=1** e carica **ECX:EBX** in **varQword**; alla fine si ottiene quindi **varQword=22FF44813DF1AAB0h** e **ZF=1**. Se **EDX:EAX** fosse stato diverso da **varQword**, la CPU avrebbe posto **ZF=0** e avrebbe caricato **DEST** in **EDX:EAX**; alla fine avremmo ottenuto quindi **EDX:EAX=3F2D40312883F890h** e **ZF=0**.
Contrariamente a quanto accade con **CMP**, l'esecuzione di **CMPXCHG8B** modifica il solo campo **ZF** del **Flags Register**.

L'istruzione CPUID.

Con il mnemonico **CPUID** si indica l'istruzione **CPU Identification** (identificazione del modello di CPU); questa istruzione che e' disponibile solo per le CPU **80586** e superiori non richiede nessun operando ma utilizza il contenuto di **EAX** come parametro. Il codice macchina di **CPUID** e' formato dagli opcodes **00001111b**, **10100010b**; quando la CPU incontra questo codice macchina restituisce nei registri generali a **32** bit una numerosissima serie di informazioni relative al modello di CPU installato nel computer. In particolare, eseguendo **CPUID** con **EAX=0** si ottiene una stringa da **12** byte contenente informazioni sul produttore della CPU; i **12** codici ASCII che formano questa stringa vengono restituiti nei **12** byte dei **3** registri **EBX**, **ECX** e **EDX**. L'ordine esatto dei registri e' rappresentato dalla sequenza **EBX**, **EDX**, **ECX**; l'ordine esatto dei codici **ASCII** e' formato dalla sequenza: primo byte di **EBX**, secondo byte di **EBX**, terzo byte di **EBX**, quarto byte di **EBX**, primo byte di **EDX**, secondo byte di **EDX** e cosi' via. Come si puo' notare, la disposizione dei codici **ASCII** segue uno schema concepito espressamente per facilitare il salvataggio di queste informazioni in una stringa; vediamo un esempio che utilizza la procedura **writeString** della libreria **EXELIB**. Prima di tutto definiamo la stringa destinata a contenere le informazioni restituite da **CPUID**:
`strBuffer db 16 dup (0)`; si suppone che il segmento dati contenente questa stringa sia referenziato da **DS**. E' importante che la stringa venga inizializzata con **0** perche' **writeString** richiede una stringa **C**; a questo punto possiamo scrivere:

```
.586                          ; set di istruzioni dell'80586
xor      eax, eax             ; eax = 0
cpuid                         ; informazioni sulla CPU
mov      di, offset strBuffer ; ds:di punta a strBuffer
mov      [di+0], ebx          ; salva il primo gruppo di 4 byte
mov      [di+4], edx          ; salva il secondo gruppo di 4 byte
mov      [di+8], ecx          ; salva il terzo gruppo di 4 byte
mov      bx, di               ; ds:bx punta a strBuffer
mov      dx, 0400h            ; riga 4, colonna 0
call     writeString          ; visualizza strBuffer
```

Generalmente, nel caso di CPU **Intel** si ottiene la stringa: **GenuineIntel**; nel caso invece di CPU **AMD** si ottiene la stringa **AuthenticAMD**. Eseguendo **CPUID** con **EAX=1** si ottengono numerose informazioni tecniche sul modello di CPU installato nel computer; tutti questi aspetti vengono illustrati in dettaglio nella sezione **Win32 Assembly**.
Per poter utilizzare l'istruzione **CPUID** e' necessaria la direttiva `.586`; se disponiamo di una CPU **80586** o superiore, ma il nostro assembler non supporta questa direttiva, possiamo ugualmente utilizzare l'istruzione **CPUID** inserendo il suo codice macchina direttamente nel programma. Al posto di **CPUID** dobbiamo scrivere quindi:
`db 00001111b, 10100010b ; cpuid`, oppure:
`dw 1010001000001111b ; cpuid`. In questo caso il secondo opcode deve trovarsi ovviamente negli **8** bit piu' significativi di questa word.
L'istruzione **CPUID** e' disponibile anche nei modelli piu' recenti dell'**80486**; l'esecuzione di **CPUID** non altera nessun campo del **Flags Register**.

L'istruzione <u>NOP</u>.

Con il mnemonico **NOP** si indica l'istruzione **No OPeration** (nessuna operazione); questa istruzione non richiede nessun operando esplicito o implicito. Il codice macchina di **NOP** e' formato dal solo opcode **10010000b**; quando la CPU incontra questo codice macchina si limita semplicemente a incrementare di **1** byte il contenuto di **IP** in modo che la coppia **CS:IP** punti all'istruzione successiva a **NOP**. Il compito dell'istruzione **NOP** consiste proprio nel non fare assolutamente niente; nonostante le apparenze, l'istruzione **NOP** torna utile in molte situazioni.

Osservando ad esempio la tabella delle istruzioni della CPU si nota che un **80486 DX** a **33** MHz esegue l'istruzione **NOP** in un solo ciclo di clock pari a:
1 / 33000000 = 0.00000003 secondi; possiamo utilizzare allora un adeguato numero di istruzioni **NOP** per generare dei ritardi di milionesimi di secondo che spesso si rendono necessari quando si programmano determinate schede hardware.

L'istruzione **NOP** viene anche utilizzata dai **debuggers**; come sappiamo questo strumento consente al programmatore di effettuare la ricerca di eventuali errori nei propri programmi. Per svolgere il proprio lavoro il debugger offre la possibilita' di inserire dei punti di interruzione (**breakpoints**) all'interno del codice del programma da analizzare; in un precedente capitolo abbiamo visto che un breakpoint e' rappresentato da un byte il cui valore **11001100b** non e' altro che il codice macchina dell'**INT 03h** chiamata proprio **trap to debugger** o **breakpoint**. Quando la CPU incontra questo codice macchina chiama l'**ISR** associata all'**INT 03h**; il debugger intercettando questa chiamata e' in grado di analizzare lo stato assunto in quel preciso istante dai vari registri della CPU. Se il programmatore vuole eliminare un breakpoint deve comunicare questa richiesta al debugger che a sua volta provvede a sostituire il codice macchina (da un byte) dell'**INT 03h** con il codice macchina (da un byte) di **NOP**; in questo modo quando la CPU incontra l'istruzione **NOP** passa direttamente all'istruzione successiva.

Anche l'assembler in molte circostanze ricorre all'istruzione **NOP**; un esempio pratico e' dato dagli effetti prodotti dalla direttiva **ALIGN** che, come sappiamo, viene utilizzata per allineare il codice e i dati di un programma. Per ottenere l'allineamento richiesto, l'assembler inserisce un adeguato numero di istruzioni **NOP**; supponiamo ad ad esempio che all'offset **0005h** del blocco codice di un programma sia presente una direttiva **ALIGN** che richiede l'allineamento della prossima istruzione ad un indirizzo multiplo di **4** byte. L'assembler incontrando questa direttiva inserisce consecutivamente tre istruzioni **NOP** che vanno ad occupare gli offset **0005h**, **0006h** e **0007h**; in questo modo la prossima istruzione si viene a trovare all'offset **0008h** che e' multiplo di **4** byte.

L'esecuzione dell'istruzione **NOP** non altera nessun campo del **Flags Register**.

Le istruzioni SETcond.

Con il mnemonico **SETcond** si indicano le istruzioni **SET byte if condition is met** (attivazione di un byte se la condizione e' verificata); questo numeroso gruppo di istruzioni, disponibili solo per le CPU **80386** e superiori, richiedono un unico operando **DEST** che deve essere di tipo **reg8/mem8**. Il codice macchina di **SETcond** e' formato da un primo opcode **00001111b**, da un secondo opcode che individua la condizione da verificare e da un terzo opcode **mod_000_r/m**; la condizione da verificare si riferisce allo stato dei vari flags. Se la condizione e' verificata la CPU carica il valore **1** in **DEST**; se la condizione non e' verificata la CPU carica il valore **0** in **DEST**.
Le istruzioni **SETcond** sono molto simili formalmente alle istruzioni **Jcond**; la differenza sostanziale sta nel fatto che le istruzioni **Jcond** eseguono il trasferimento del controllo in base allo stato dei vari flags, mentre le istruzioni **SETcond** modificano l'operando **DEST** in base allo stato dei vari flags. La conseguenza pratica di tutto cio' e' che anche le istruzioni **SETcond** possono essere suddivise in tre gruppi; vediamo l'elenco completo di queste istruzioni cominciando da quelle che fanno esplicito riferimento ai flags e che valgono quindi sia per i numeri senza segno che per i numeri con segno:

istruzione	condizione	effetto
SETO	OF = 1 ?	Se (OF = 1) DEST = 1, altrimenti DEST = 0
SETC	CF = 1 ?	Se (CF = 1) DEST = 1, altrimenti DEST = 0
SETZ/SETE	ZF = 1 ?	Se (ZF = 1) DEST = 1, altrimenti DEST = 0
SETS	SF = 1 ?	Se (SF = 1) DEST = 1, altrimenti DEST = 0
SETP/SETPE	PF = 1 ?	Se (PF = 1) DEST = 1, altrimenti DEST = 0
SETNO	OF = 0 ?	Se (OF = 0) DEST = 1, altrimenti DEST = 0
SETNC	CF = 0 ?	Se (CF = 0) DEST = 1, altrimenti DEST = 0
SETNZ/SETNE	ZF = 0 ?	Se (ZF = 0) DEST = 1, altrimenti DEST = 0
SETNS	SF = 0 ?	Se (SF = 0) DEST = 1, altrimenti DEST = 0
SETNP/SETPO	PF = 0 ?	Se (PF = 0) DEST = 1, altrimenti DEST = 0

Il secondo gruppo di istruzioni **SETcond** fa riferimento alle modifiche subite dai flags in seguito ad una operazione logico aritmetica che ha coinvolto due numeri senza segno; nella tabella i due numeri vengono indicati con **op1** e **op2**

Istruzioni SETcond per i numeri senza segno		
istruzione	**condizione**	**effetto**
SETB/SETNAE	(op1 < op2) ?	se (op1 < op2) DEST = 1, altrimenti DEST = 0
SETNB/SETAE	(op1 >= op2) ?	se (op1 >= op2) DEST = 1, altrimenti DEST = 0
SETBE/SETNA	(op1 <= op2) ?	se (op1 <= op2) DEST = 1, altrimenti DEST = 0
SETNBE/SETA	(op1 > op2) ?	se (op1 > op2) DEST = 1, altrimenti DEST = 0

Il terzo gruppo di istruzioni **SETcond** fa riferimento alle modifiche subite dai flags in seguito ad una operazione logico aritmetica che ha coinvolto due numeri con segno; nella tabella i due numeri vengono indicati con **op1** e **op2**

Istruzioni SETcond per i numeri con segno		
istruzione	**condizione**	**effetto**
SETL/SETNGE	(op1 < op2) ?	se (op1 < op2) DEST = 1, altrimenti DEST = 0
SETNL/SETGE	(op1 >= op2) ?	se (op1 >= op2) DEST = 1, altrimenti DEST = 0
SETLE/SETNG	(op1 <= op2) ?	se (op1 <= op2) DEST = 1, altrimenti DEST = 0
SETNLE/SETG	(op1 > op2) ?	se (op1 > op2) DEST = 1, altrimenti DEST = 0

Per quanto riguarda il doppio nome utilizzato da molti mnemonici valgono tutte le considerazioni gia' esposte per le istruzioni **Jcond**.
L'utilizzo delle istruzioni **SETcond** e' abbastanza semplice; possiamo scrivere ad esempio:

```
test     ax, ax      ; ax = 0 ?
setz     bl          ; se ax = 0, bl = 1
```

Se **AX=0** l'istruzione **TEST** produce un risultato nullo (**ZF=1**) e si ottiene quindi **BL=1**; se invece **AX** e' diverso da zero l'istruzione **TEST** produce un risultato non nullo (**ZF=0**) e si ottiene quindi **BL=0**
L'esecuzione delle istruzioni **SETcond** non altera nessun campo del **Flags Register**.

L'istruzione <u>XADD</u>.

Con il mnemonico **XADD** si indica l'istruzione **eXchange and ADD** (scambia e somma due operandi); questa istruzione, che e' disponibile solo per le CPU **80486** e superiori, richiede esplicitamente due operandi che svolgono il ruolo di **SRC** e **DEST**. L'operando **DEST** deve essere di tipo **reg/mem**; l'operando **SRC** deve essere di tipo **reg**. Il codice macchina dell'istruzione **XADD** e' formato dagli opcodes **00001111b**, **1100000_w** e **mod_reg_r/m**; quando la CPU incontra questo codice macchina, trasferisce **DEST** in **SRC**, somma **DEST** con il vecchio contenuto di **SRC** e trasferisce il risultato della somma in **DEST**.
Vediamo un esempio:

```
.486                 ; set di istruzioni dell'80486
mov      ax, 3500    ; DEST
mov      bx, 8000    ; SRC
xadd     ax, bx
```

L'istruzione **XADD** trasferisce in **BX** il contenuto (**3500**) di **AX**, somma il contenuto (**3500**) di **AX** con il vecchio contenuto (**8000**) di **BX** e trasferisce il risultato (**11500**) in **AX**; alla fine si ottiene **AX=11500** e **BX=3500**.
Come accade per l'istruzione **ADD**, anche l'esecuzione dell'istruzione **XADD** modifica i campi campi **OF**, **SF**, **ZF**, **AF**, **PF**, **CF** del **Flags Register**.

Le istruzioni XLAT, XLATB.

Con il mnemonico **XLAT** si indica l'istruzione **table look-up translation** (conversione attraverso una tabella di consultazione); questa istruzione richiede esplicitamete un solo operando di tipo **mem8** al quale si accede obbligatoriamente attraverso il registro puntatore **BX**. In assenza del segment override la CPU utilizza **DS** come registro di segmento predefinito; e' consentito il segment override che permette di utilizzare in coppia con **BX** qualsiasi registro di segmento tra **CS**, **SS**, **DS**, **ES** (piu' **FS** e **GS** per le CPU **80386** e superiori). Il registro **BX** punta ad una **look-up table** (tabella di consultazione) che si trova in memoria; in generale una **look-up table** e' una tabella contenente una serie di dati numerici. Ogni volta che un programma ha bisogno di uno di questi dati lo puo' leggere dalla tabella specificando le coordinate (riga, colonna) associate al dato stesso; una tabella formata da una sola riga, assume la struttura di un vettore di **n** elementi dove ogni elemento e' rappresentato da un indice. Nel caso dell'istruzione **XLAT**, la tabella di consultazione deve avere la struttura di un vettore di byte; come indice del vettore (**0**, **1**, **2**, etc) la CPU utilizza implicitamente il registro **AL**. Il codice macchina dell'istruzione **XLAT** e' formato dal solo opcode **11010111b**; quando la CPU incontra questo codice macchina, accede alla locazione di memoria che si trova all'offset **BX+AL** legge il contenuto a **8** bit e lo carica in **AL**.
AL ha un'ampiezza di **8** bit per cui con questo registro possiamo specificare un indice compreso tra **0** e **255**; in sostanza, possiamo creare una **look-up table** formata al massimo da **256** elementi da **1** byte ciascuno. Come si puo' facilmente intuire, le caratteristiche dell'istruzione **XLAT** si prestano molto bene per gestire una **look-up table** contenente i **256** simboli del codice ASCII; infatti l'istruzione **XLAT** e' stata creata dalla **Intel** proprio per effettuare conversioni tra codice **ASCII** e codici utilizzati su altre piattaforme hardware. In particolare, l'istruzione **XLAT** viene impiegata in modo massiccio per effettuare conversioni tra codice **ASCII** e codice **EBCDIC** (**Extended Binary Coded Decimal Interchange Code**); il codice **EBCDIC** viene utilizzato dai **mainframe** della **IBM**. Se vogliamo far comunicare un **mainframe** con un **PC** dobbiamo scrivere quindi un apposito programma di conversione da **ASCII** a **EBCDIC** e viceversa; a titolo di curiosita' vediamo in questa tabella i codici **EBCDIC** delle lettere minuscole dell'alfabeto:

Codici EBCDIC delle lettere minuscole		
129 = a	145 = j	162 = s
130 = b	146 = k	163 = t
131 = c	147 = l	164 = u
132 = d	148 = m	165 = v
133 = e	149 = n	166 = w
134 = f	150 = o	167 = x
135 = g	151 = p	168 = y
136 = h	152 = q	169 = z
137 = i	153 = r	_______

Come si puo' notare, a differenza di quanto accade con il codice **ASCII**, i codici **EBCDIC** delle lettere minuscole sono consecutivi ma non contigui; questo significa che non e' possibile applicare al codice **EBCDIC** gli algoritmi che vengono utilizzati usualmente per il codice **ASCII**.
Per illustrare il funzionamento dell'istruzione **XLAT** vediamo proprio un esempio relativo alla conversione da **ASCII** a **EBCDIC**; in questo esempio creiamo una **look-up table** che riceve in input un codice **ASCII** di una lettera minuscola e restituisce in output il corrispondente codice **EBCDIC**. Nel codice **ASCII** le lettere minuscole occupano le posizioni consecutive e contigue comprese tra **97** e **122**; nel blocco dati del programma definiamo quindi la seguente tabella:

```
tabEBCDIC      db    129, 130, 131, 132, 133, 134, 135, 136, 137
               db    145, 146, 147, 148, 149, 150, 151, 152, 153
               db    162, 163, 164, 165, 166, 167, 168, 169
```

Supponendo che questo blocco dati venga referenziato da **DS**, la tabella **tabEBCDIC** verra' puntata da **DS:BX**; la conversione da **ASCII** a **EBCDIC** viene naturalmente effettuata da **XLAT** che si aspetta di trovare in **AL** l'indice di **tabEBCDIC** a cui accedere. Gli indici della tabella partono da zero mentre i codici **ASCII** delle lettere minuscole partono da **97**; questo significa che se vogliamo conoscere ad esempio il codice **EBCDIC** della lettera **'f'** dobbiamo caricare in **AL** l'indice:
'f' - 97 e cioe' **102 - 97 = 5**. L'elemento in posizione **5** all'interno di **tabEBCDIC** vale **134** che e' proprio il codice **EBCDIC** della lettera **'f'**; a questo punto possiamo scrivere le seguenti istruzioni:

```
mov      bx, offset tabEBCDIC    ; ds:bx punta a tabEBCDIC
mov      al, 'f' - 97            ; indice tabella
xlat     [bx]                    ; al = ds:[bx+5] = 134
```

L'istruzione **XLAT** accede all'indice **5** della tabella **tabEBCDIC**, legge il valore **134** e lo carica in **AL**; in modo analogo, caricando in **AL** il valore
'z' - 97 = 122 - 97 = 25 ed eseguendo **XLAT** otteniamo in **AL** il valore **169** che occupa infatti la posizione **25** in **tabEBCDIC** e rappresenta proprio il codice **EBCDIC** della lettera **'z'**.
Se vogliamo effettuare conversioni da **EBCDIC** a **ASCII** dobbiamo seguire lo stesso procedimento; naturalmente in questo caso bisogna ricordarsi di inserire nella **look-up table** anche gli indici che in **EBCDIC** non vengono utilizzati, come ad esempio tutti gli indici compresi tra **138** e **144**.
L'esempio mostrato in precedenza e' piuttosto semplice e non evidenzia le potenzialita' di **XLAT**; nei programmi reali in genere si utilizza un loop all'interno del quale si sviluppa un flusso di byte che vengono letti in sequenza da un buffer, convertiti con **XLAT** e inviati via modem a un'altro computer.

Se non si ha bisogno del segment override si puo' utilizzare direttamente l'istruzione **XLATB**; questa istruzione non richiede nessun operando in quanto utilizza implicitamente (ed esclusivamente) **DS:BX** e **AL**.
L'esecuzione delle istruzioni **XLAT/XLATB** non altera nessun campo del **Flags Register**.

Capitolo 24 - Operatori, direttive e macro

Gli assembler piu' sofisticati come **MASM** e **TASM** mettono a disposizione una serie di potenti strumenti che spesso si rivelano di grande aiuto per il programmatore; questi strumenti possono essere suddivisi in tre grandi categorie chiamate: **operatori**, **direttive** e **macro**. Gli operatori ci permettono di creare complesse espressioni matematiche da inserire all'interno di un programma Assembly; e' possibile costruire espressioni valutabili in fase di assemblaggio o in fase di esecuzione di un programma. Le direttive hanno principalmente lo scopo di permettere al programmatore di impartire ordini all'assembler modificandone la modalita' operativa; in questo capitolo esamineremo in particolare le direttive condizionali che ci permettono di creare costrutti sintattici simili a quelli dei linguaggi di alto livello. Attraverso le macro e' possibile rappresentare con un nome simbolico intere porzioni di un programma Assembly; in questo capitolo vengono anche illustrate le macro predefinite messe a disposizione da **MASM** e **TASM**.

Gli operatori per la fase di assemblaggio.

Cominciamo con una serie di operatori che vengono chiamati **assembler time operators** (operatori per la fase di assemblaggio); questa definizione e' legata al fatto che generalmente questi operatori compaiono in espressioni che devono poter essere valutate dall'assembler in fase di assemblaggio del programma. In sostanza l'assembler deve avere la possibilita' di analizzare e risolvere queste espressioni, ricavandone alla fine un valore numerico (risultato dell'espressione) che verra' inserito direttamente nel codice macchina; proprio per questo motivo, gli **assembler time operators** possono comparire solo in espressioni comprendenti operandi immediati. Non e' possibile quindi utilizzare operandi di tipo **reg** o **mem**; e' chiaro infatti che l'assembler non puo' conoscere in anticipo il contenuto che verra' assunto da un registro o da una variabile in fase di esecuzione del programma.
Il modo piu' semplice per creare un operando immediato consiste nell'utilizzare la direttiva di assegnamento = (uguale); la sintassi generica per questa direttiva e' la seguente:
`nome = espressione.`
In questo caso con il termine **espressione** si indica un valore immediato o una qualunque espressione matematica formata da operatori dell'Assembly e da operandi immediati. Nel caso piu' semplice possiamo scrivere ad esempio:
`LUNGHEZZA = 3500;`
in questo caso **LUNGHEZZA** e' il nome simbolico dell'operando immediato, mentre **3500** e' il valore immediato che la direttiva = assegna all'operando stesso. Il nome simbolico **LUNGHEZZA** puo' essere utilizzato in qualunque punto del programma; ogni volta che l'assembler incontra questo nome, provvede a sostituirlo con il valore **3500**.
Al posto di = si puo' anche utilizzare la direttiva **EQU**; questa direttiva viene descritta piu' avanti. La differenza sostanziale che esiste tra = e **EQU** sta nel fatto che = permette solo di assegnare valori numerici ad un nome simbolico; la direttiva **EQU** invece e' molto piu' potente in quanto permette di rappresentare con un nome simbolico sia valori numerici che stringhe alfanumeriche. Un nome simbolico al quale e' stato assegnato un valore con = puo' essere ridichiarato piu' volte (sempre con =) in qualunque altro punto del programma; nel caso di **EQU** invece e' possibile ridichiarare solo nomi simbolici che rappresentano stringhe. Nel caso in cui si vogliano dichiarare operandi di tipo numerico si raccomanda di utilizzare sempre la direttiva =.
Tutti gli operatori dell'assembler lavorano su operandi immediati di tipo intero; questo significa che non sono consentite dichiarazioni di operandi immediati del tipo:
`PI_GRECO = 3.14.`
Le ampiezze in bit degli operandi non possono superare le capacita' della CPU che si sta utilizzando; non e' possibile quindi dichiarare operandi a **32** bit con una CPU a **16** bit. Per quanto riguarda la distinzione tra numeri con o senza segno, valgono tutte le considerazioni gia' esposte nei

precedenti capitoli in relazione agli operandi di tipo **reg** e **mem**; e' compito del programmatore quindi decidere se i vari operandi di tipo **imm** debbano essere trattati come numeri con o senza segno (piu' avanti vengono esposti ulteriori chiarimenti su questo argomento).
Analizziamo in dettaglio i vari gruppi di operatori cominciando da quelli aritmetici; nelle tabelle che seguono, i termini **expr1**, **expr2**, **expr3**, etc, indicano generiche espressioni che l'assembler deve poter risolvere.

Operatori aritmetici			
operatore	**significato**	**esempio**	**effetto**
+	piu' unario	+expr1	lascia invariato il segno di expr1
+	piu' binario	expr1 + expr2	somma expr1 con expr2
-	meno unario	-expr1	cambia il segno di expr1
-	meno binario	expr1 - expr2	sottrae expr2 da expr1
*	moltiplicazione	expr1 * expr2	moltiplica expr1 con expr2
/	divisione intera	expr1 / expr2	divide expr1 per expr2
MOD	operatore modulo	expr1 MOD expr2	resto di expr1 / expr2

Vediamo alcuni esempi pratici che illustrano il funzionamento di questi operatori; supponiamo di avere le seguenti dichiarazioni:

```
OPERANDO1   = +12000
OPERANDO2   = +2
OPERANDO3   = -2
EXPR1       = (OPERANDO1 + OPERANDO2) - OPERANDO3
EXPR2       = (OPERANDO2 - OPERANDO1) - OPERANDO3
EXPR3       = OPERANDO3 - OPERANDO1
EXPR4       = (OPERANDO1 * OPERANDO3) / OPERANDO2
EXPR5       = ((OPERANDO1 - OPERANDO2) * OPERANDO3) / OPERANDO2
EXPR6       = -(OPERANDO1 * OPERANDO3)
```

Quando l'assembler incontra queste dichiarazioni cerca di analizzarle e di risolverle in modo da assegnare a ciascuna di esse un ben preciso valore numerico; in seguito, ogni volta che l'assembler incontra uno di questi nomi, lo sostituisce con il corrispondente valore numerico. Al nome simbolico **OPERANDO1** l'assembler assegna il valore numerico positivo **+12000**; al nome simbolico **OPERANDO2** viene assegnato il valore numerico positivo **+2**. Al nome simbolico **OPERANDO3** viene assegnato il valore numerico negativo **-2**; utilizzando gli operandi immediati **OPERANDO1**, **OPERANDO2** e **OPERANDO3** (che devono essere gia' stati dichiarati), possiamo costruire le espressioni **EXPR1**, **EXPR2**, **EXPR3**, **EXPR4**, **EXPR5** e **EXPR6** mostrate nella tabella precedente. Per assegnare un valore numerico ai nomi simbolici che rappresentano queste espressioni, l'assembler deve effettuare una serie di calcoli. Come si puo' notare, e' consentito l'uso delle parentesi tonde che permettono di stabilire l'ordine di valutazione delle espressioni; le parentesi tonde permettono anche di alterare l'ordine di precedenza degli operatori che verra' illustrato piu' avanti.
Per **EXPR1** l'assembler calcola:
`(12000 + 2) - (-2) = 12002 + 2 = 12004`;
alla fine del calcolo l'assembler assegna a **EXPR1** il numero positivo **+12004**.
Per **EXPR2** l'assembler calcola:
`(2 - 12000) - (-2) = -11998 + 2 = -11996`;
alla fine del calcolo l'assembler assegna a **EXPR2** il numero negativo **-11996**.
Per **EXPR3** l'assembler calcola:

```
-2 - 12000 = -12002;
```

alla fine del calcolo l'assembler assegna a **EXPR3** il numero negativo **-12002**.
Per **EXPR4** l'assembler calcola:

```
(12000 * (-2)) / 2 = -24000 / 2 = -12000;
```

alla fine del calcolo l'assembler assegna a **EXPR4** il numero negativo **-12000**.
Per **EXPR5** l'assembler calcola:

```
((12000 - 2) * (-2)) / 2 = (11998 * (-2)) / 2 = -23996 / 2 = -11998;
```

alla fine del calcolo l'assembler assegna a **EXPR5** il numero negativo **-11998**.
Per **EXPR6** l'assembler calcola:

```
-(12000 * (-2)) = -(-24000) = +24000;
```

alla fine del calcolo l'assembler assegna a **EXPR6** il numero positivo **+24000**.
Tenendo conto del fatto che stiamo operando su numeri interi, e' chiaro che l'operatore **/** si riferisce alla divisione intera che provoca il troncamento di tutte le cifre che nel risultato (quoziente) figurano dopo la virgola; nel caso ad esempio di:

```
12500 / 3,
```

si ottiene **4166** con troncamento della parte decimale. Analogamente, l'operatore **MOD** (modulo) restituisce il resto della divisione intera tra due operandi.
Tornando alla delicata questione del segno, possiamo dire che nel caso in cui nell'espressione figurino solo addizioni e sottrazioni, si ottiene un risultato che e' valido sia per i numeri con segno che per quelli senza segno; nel caso invece di moltiplicazioni e/o divisioni che come sappiamo provocano un cambiamento di modulo, l'assembler segue un comportamento ben preciso:
a) se nell'espressione compaiono solo operandi positivi, viene prodotto un risultato che appartiene all'insieme dei numeri senza segno;
b) se nell'espressione compare almeno un operando negativo (cioe' dichiarato con il segno meno davanti), viene prodotto un risultato in complemento a due che appartiene all'insiemne dei numeri con segno.

Le espressioni del tipo appena illustrato, possono essere inserite anche all'interno delle istruzioni Assembly; naturalmente, anche in questo caso l'assembler deve essere in grado di convertire queste espressioni in valori immediati. Supponiamo ad esempio che **Vector1** rappresenti un vettore di **10** word; ciascun elemento di **Vector1** occupa quindi **2** byte. Se vogliamo caricare in **BX** l'offset del sesto elemento (indice **5**) di **Vector1**, possiamo scrivere:

```
mov bx, offset Vector1 + (2 * 5),
```

oppure

```
mov bx, offset Vector1[2 * 5].
```

Ricordiamo che **OFFSET** e' un'operatore dell'Assembly che applicato al nome di una variabile, ne restituisce il relativo offset (che e' un valore immediato); se ad esempio **Vector1** si trova all'offset **00B4h** del blocco dati, l'assembler calcola:

```
00B4h + (2 * 5) = 00B4h + 000Ah = 00BEh,
```

e genera quindi il codice macchina dell'istruzione che carica il valore immediato **00BEh** in **BX**.
Un'altro esempio puo' essere rappresentato da:

```
mov ax, ((OPERANDO1 - OPERANDO2) * OPERANDO3) / OPERANDO2;
```

in base ai valori assegnati in precedenza a questi tre operandi, possiamo dire che l'assembler produrra' il codice macchina dell'istruzione che carica in **AX** il numero negativo **-11998**.
Possiamo utilizzare espressioni di qualunque complessita' senza il rischio di provocare ripercussioni negative sul programma (in termini di prestazioni o di dimensioni del codice); ricordiamoci infatti che tutti i calcoli vengono svolti dall'assembler che in fase di assemblaggio del programma provvede a risolvere queste espressioni sostituendole con il corrispondente risultato (valore immediato). Nel codice macchina del programma quindi, non rimane nessuna traccia di tutti questi calcoli; come al solito, se vogliamo verificare il lavoro svolto dall'assembler, ci basta richiedere la generazione del listing file.

La tabella seguente mostra gli operatori logici messi a disposizione da **MASM** e **TASM**; naturalmente, questi operatori non devono essere confusi con le omonime istruzioni Assembly.

Operatori logici bit a bit		
operatore	**significato**	**esempio**
OR	OR logico inclusivo	expr1 OR expr 2
XOR	OR logico esclusivo	expr1 XOR expr2
AND	AND logico	expr1 AND expr2
NOT	complemento a uno (inversione)	NOT expr1

Cosi' come accade per le istruzioni omonime, anche gli operatori logici dell'assembler agiscono sui singoli bit dei loro operandi immediati; questo significa che se abbiamo ad esempio le seguenti dichiarazioni:

```
VAL1 = 10101010b
VAL2 = 01010101b
VAL3 = 00001111b
```

allora:
```
EXPR1 = NOT VAL1
```
assegna a **EXPR1** il numero binario **01010101b**;
```
EXPR2 = VAL1 OR VAL2
```
assegna a **EXPR2** il numero binario **11111111b**;
```
EXPR3 = VAL1 XOR VAL3
```
assegna a **EXPR3** il numero binario **10100101b**;
```
EXPR4 = VAL2 AND VAL3
```
assegna a **EXPR4** il numero binario **00001010b**;
```
mov ax, VAL1 AND (NOT VAL3)
```
fa generare all'assembler il codice macchina dell'istruzione che carica in **AX** il numero binario **10100000b**; questo numero prima di essere caricato in **AX** viene esteso con degli zeri (zero extended) a **16** bit. Infatti gli operatori **VAL1**, **VAL2** e **VAL3** sono stati dichiarati senza segno **-** davanti per cui vengono trattati dall'assembler come numeri senza segno.

La tabella seguente mostra gli operatori di scorrimento dei bit messi a disposizione da **MASM** e **TASM**; anche in questo caso non bisogna confondere questi operatori con le istruzioni omonime dell'Assembly:

Operatori di scorrimento dei bit		
operatore	**significato**	**esempio**
SHL	shift logical left	expr1 SHL contatore
SHR	shift logical right	expr1 SHR contatore

Questi due operatori trattano il loro operando di sinistra come un numero senza segno (scorrimento logico dei bit); non esistono quindi operatori per lo scorrimento aritmetico dei bit, che tengono cioe' conto del bit di segno dell'operando. Nello scorrimento a destra, i bit dell'operando che traboccano da destra vengono persi; nello scorrimento a sinistra, l'assembler finche' e' possibile cerca di estendere l'ampiezza in bit del risultato. Se abbiamo ad esempio la seguente dichiarazione:
```
VAL1 = 10101010b,
```
allora l'espressione:

```
EXPR1 = VAL1 SHL 1,
```

assegna a **EXPR1** il numero binario a **16** bit **0000000101010100b**; come si puo' notare, l'assembler tratta **EXPR1** come una costante a **16** bit in modo da farle contenere interamente il risultato prodotto da **SHL**. E' chiaro che tutto cio' ha un limite rappresentato dall'ampiezza massima in bit gestibile dalla CPU che stiamo utilizzando; se abbiamo ad esempio una CPU a **16** bit e scriviamo:

```
VAL1 = 1000000000000000b
EXPR1 = VAL1 SHL 1

   mov      ax, EXPR1
```

verra' caricato in **AX** il valore zero. Infatti **EXPR1** non puo' avere un'ampiezza superiore a **16** bit e quindi non e' in grado di contenere il risultato completo a **17** bit dello scorrimento a sinistra; il bit piu' significativo del risultato trabocca da sinistra e viene perso, per cui a **EXPR1** verra' assegnato il numero binario a **16** bit **0000000000000000b**.
Un'aspetto curioso degli operatori **SHL** e **SHR**, e' dato dal fatto che il contatore che indica il numero di scorrimenti di bit da effettuare, puo' essere anche un numero negativo; in questo caso si produce l'effetto di far scorrere i bit nel verso opposto a quello indicato dall'operatore. Se scriviamo ad esempio:

```
VAL1 = 10101010b
EXPR1 = VAL1 SHL -1
```

otteniamo l'assegnamento a **EXPR1** del numero binario **01010101b**; come si puo' notare, il contatore negativo ha fatto scorrere i bit di **VAL1** verso destra anziche' verso sinistra.

La tabella seguente mostra gli operatori relazionali messi a disposizione da **MASM** e **TASM**; attraverso questi operatori e' possibile confrontare tra loro operandi immediati o espressioni complesse contenenti operandi immediati:

Operatori relazionali		
operatore	**significato**	**esempio**
EQ	equal (uguale a)	expr1 EQ expr2
NE	not equal (diverso da)	expr1 NE expr2
LT	less than (minore di)	expr1 LT expr2
LE	less than or equal (minore o uguale a)	expr1 LE expr2
GT	greater than (maggiore di)	expr1 GT expr2
GE	greater than or equal (maggiore o uguale a)	expr1 GE expr2

Combinando tra loro due o piu' espressioni attraverso gli operatori relazionali, si ottiene una cosiddetta **espressione booleana**; come gia' sappiamo il risultato di una espressione booleana puo' essere vero o falso (**TRUE** o **FALSE**). Convenzionalmente, in tutti i linguaggi di programmazione (compreso l'Assembly), un risultato **FALSE** viene associato al valore zero; un risultato **TRUE** viene invece associato ad un valore diverso da zero (valore non nullo) stabilito dal particolare compilatore o assemblatore che si sta usando. Il programmatore deve solo ricordarsi di questa regola e non ha la necessita' di conoscere il valore numerico esatto associato ad un risultato **TRUE**; si puo' dire anzi che e' un grave errore scrivere codice che fa esplicito riferimento al valore esatto assegnato ad un risultato **TRUE**.
Supponiamo di avere:

```
VAL1 = 3500
VAL2 = 8000

EXPR1 = VAL1 EQ VAL2
EXPR2 = VAL1 LT VAL2
EXPR3 = VAL1 GT VAL2
```

In questo caso, tenendo conto del fatto che **VAL1** e' chiaramente minore (**LT**) di **VAL2**, si ottiene l'assegnamento di un valore nullo (zero) e **EXPR1** e **EXPR3**, mentre a **EXPR2** verra' assegnato un valore diverso da zero.
Gli operatori relazionali esprimono tutta la loro potenza quando vengono usati in combinazione con le direttive condizionali **IF**, **ESLSEIF**, etc; questi aspetti vengono illustrati piu' avanti.

In questo paragrafo sono stati illustrati solo alcuni dei numerosissimi operatori messi a disposizione da **MASM** e **TASM**; nei precedenti capitoli abbiamo gia' fatto conoscenza con diversi operatori dell'Assembly come **SEG**, **OFFSET**, **BYTE PTR**, **WORD PTR**, etc. Altri operatori importanti verranno illustrati nei prossimi capitoli; per ulteriori dettagli si consiglia di consultare i manuali di **MASM** e **TASM** reperibili anche attraverso Internet.

In conclusione di questo paragrafo analizziamo la tabella relativa all'ordine di precedenza dei vari operatori dell'Assembly; gli operatori appartenenti alla stessa riga hanno uguale precedenza, mentre le varie righe indicano i diversi ordini di precedenza, dal piu' alto (riga **1**) al piu' basso (riga **13**).

Ordine di precedenza degli operatori	
ordine	**operatori**
1	(), [], LENGTH, MASK, SIZE, WIDTH
2	. (selettore del membro di una struttura)
3	HIGH, LOW
4	+ (unario), - (unario)
5	: (segment override)
6	OFFSET, PTR, SEG, THIS, TYPE
7	*, /, MOD, SHL, SHR
8	+ (binario), - (binario)
9	EQ, GE, GT, LE, LT, NE
10	NOT
11	AND
12	OR, XOR
13	LARGE, SHORT, SMALL, .TYPE

I diversi ordini di precedenza dei vari operatori, permettono all'assembler di valutare in modo corretto istruzioni del tipo:

```
mov bx, offset Vector1 + (2 * 5).
```

Siccome l'operatore **OFFSET** ha precedenza maggiore dell'operatore **+** binario, l'assembler determina innanzi tutto l'offset di **Vector1**, poi calcola **2 * 5 = 10** e infine somma il valore **10** all'offset di **Vector1**; se **+** avesse avuto precedenza maggiore di **OFFSET** avremmo ottenuto un'istruzione senza senso indicante all'assembler di sommare **10** a **Vector1** e calcolare poi l'offset del risultato della somma.

L'ordine di precedenza dei vari operatori e' una caratteristica importantissima di tutti i linguaggi di programmazione che forniscono sempre un'apposita tabella relativa a questo aspetto; in ogni caso, un programmatore avveduto e responsabile non dovrebbe fidarsi ciecamente di queste tabelle. Un compilatore (o un assemblatore) che non gestisce correttamente gli ordini di precedenza dei vari operatori, puo' generare programmi contenenti dei bugs difficilissimi da scovare; in caso di dubbio si consiglia vivamente di ricorrere alle parentesi tonde che permettono di specificare in modo esplicito l'ordine di valutazione di un'espressione.

Gli operatori per la fase di esecuzione.

Le versioni piu' recenti di **MASM** e **TASM** forniscono una serie di potenti operatori utilizzabili in fase di esecuzione di un programma; per questo motivo si parla anche di **run time operators** (operatori per la fase di esecuzione). Tutti questi operatori risultano molto familiari ai programmatori **C** in quanto utilizzano la classica sintassi degli operatori relazionali e logici del linguaggio **C**; la tabella seguente mostra i run time operators piu' importanti:

Run time operators		
operatore	**significato**	**esempio**
==	uguale a	expr1 == expr2
!=	diverso da	expr1 != expr2
<	minore di	expr1 < expr2
<=	minore o uguale a	expr1 <= expr2
>	maggiore di	expr1 > expr2
>=	maggiore o uguale a	expr1 >= expr2
||	OR logico	expr1 || expr2
&&	AND logico	expr1 && expr2
&	AND logico bit-a-bit	expr1 & expr2
!	negazione logica	!expr1

Anche in questo caso quindi, abbiamo a che fare con una serie di operatori relazionali e logici; la differenza sostanziale rispetto agli assembler time operators e' data dal fatto che con i run time operators possiamo utilizzare non solo operandi di tipo **imm**, ma anche operandi di tipo **reg** e **mem**!
I run time operators possono essere utilizzati esclusivamente in combinazione con una serie di apposite direttive per il controllo del flusso; queste direttive vengono illustrate piu' avanti. L'uso dei run time operators di tipo relazionale e' abbastanza intuitivo; ad esempio, l'espressione:

```
(ax == bx)
```

restituisce **TRUE** solo se il contenuto di **AX** coincide con il contenuto di **BX**. Analogamente, l'espressione:

```
(ax < bx)
```

restituisce **TRUE** solo se il contenuto di **AX** e' minore del contenuto di **BX**.
Per quanto riguarda i run time operators di tipo logico e' necessario fornire alcuni chiarimenti; come sa chi utilizza i linguaggi di programmazione di alto livello, e' necessario distinguere tra operatori logici bit-a-bit e operatori logici che agiscono sul valore numerico contenuto negli operandi.
Consideriamo ad esempio un'espressione del tipo:

```
(ax & bx);
```

in questo caso viene eseguito un **AND** bit-a-bit tra **AX** e **BX** che, come gia' sappiamo, consiste nel confrontare il bit in posizione **0** di **AX** con il bit in posizione **0** di **BX**, il bit in posizione **1** di **AX** con il bit in posizione **1** di **BX**, etc. Alla fine si ottiene un valore a **16** bit che rappresenta il risultato

dei **16** confronti bit-a-bit appena effettuati; se il risultato ottenuto e' diverso da zero, l'espressione vale **TRUE**. Consideriamo invece la seguente espressione:

```
(ax && bx);
```

in questo caso viene effettuato un **AND** tra il contenuto di **AX** e il contenuto di **BX**. Il risultato che si ottiene vale **TRUE** solo se entrambi gli operandi sono diversi da zero; se almeno un operando vale zero si ottiene un risultato **FALSE**.
Lo stesso discorso vale per l'**OR** logico (||) che restituisce **TRUE** solo se almento uno dei suoi operandi e' diverso da zero; come si puo' notare, non esiste ne l'**OR** logico bit-a-bit ne l'operatore **XOR**.
L'operatore di negazione logica (**!**) restituisce **TRUE** se il suo operando vale zero; se invece l'operando **op** e' diverso da zero allora **!op** restituisce **FALSE**.

Un'ultima considerazione da fare riguarda gli assembler time operators **+** binario, **-** binario e ***** binario; come abbiamo gia' visto nei precedenti capitoli, questi tre operatori possono essere utilizzati anche a run time per gestire gli indirizzamenti. Con tutte le CPU della famiglia **Intel** si puo' scrivere ad esempio:

```
mov ax, [di+2];
```

questa istruzione trasferisce in **AX** un valore a **16** bit che si trova all'offset **DI+2**. Analogamente l'istruzione:

```
mov ax, [di-2]
```

trasferisce in **AX** un valore a **16** bit che si trova all'offset **DI-2**.
Con le CPU **80386** e superiori e' inoltre possibile utilizzare l'operatore ***** per gestire il nuovo formato di indirizzamento **SIB** (Scale Index Base); usando ad esempio **EBP** come registro base, **EDI** come registro indice, **4** come fattore di scala e il valore immediato **DISP32** come spiazzamento, possiamo scrivere l'istruzione:

```
mov eax, [ebp + (edi * 4) + DISP32].
```

Tutti questi calcoli verranno eseguiti dalla CPU in fase di esecuzione del programma.

Le direttive condizionali per la fase di assemblaggio.

Nei precedenti capitoli abbiamo gia' incontrato numerose direttive dell'Assembly; possiamo citare ad esempio **ASSUME**, **SEGMENT**, **ALIGN**, **END**, **INCLUDE**, **.8086**, **.386**, **DB**, **DW**, etc. Altre direttive verranno illustrate al momento opportuno nei capitoli successivi; in questo capitolo invece vengono illustrate in particolare le direttive condizionali che vengono usate in combinazione con gli operatori descritti in precedenza.
In analogia a quanto accade per gli operatori, anche le direttive si suddividono in **assembler time directives** e **run time directives**; partiamo dalle direttive per la fase di assemblaggio che indubbiamente sono le piu' utili (e le piu' sensate). Il primo gruppo di assembler time directives che andiamo ad esaminare comprende le direttive **IF**, **ELSEIF**, **ELSE** e **ENDIF**; queste direttive ci permettono di effettuare una scelta tra piu' opzioni possibili. Vediamo un esempio pratico che illustra l'utilizzo abbastanza intuitivo di queste direttive; supponiamo di avere le seguenti dichiarazioni:

```
VAL1 = 3500
VAL2 = 8000
```

A questo punto, nel blocco codice del nostro programma possiamo scrivere:

```
IF (VAL1 LT VAL2)        ; se VAL1 e' minore di  VAL2
   mov      ax, 10
ELSEIF (VAL1 EQ VAL2)    ; se VAL1 e' uguale a VAL2
   mov      ax, 20
ELSE                     ; se VAL1 e' maggiore di VAL2
   mov      ax, 30
ENDIF
```

Questa sequenza dice all'assembler che se la prima espressione e' verificata bisogna assemblare la prima istruzione, altrimenti, se la seconda espressione e' verificata bisogna assemblare la seconda istruzione; se nessuna delle due precedenti espressioni e' verificata bisogna assemblare la terza istruzione. In fase di assemblaggio del programma, l'assembler rileva che **VAL1** e' minore di **VAL2** per cui solo la prima espressione booleana e' **TRUE**; la conseguenza pratica e' data dal fatto che verra' assemblata esclusivamente l'istruzione:

`mov ax, 10.`

Per verificarlo possiamo consultare il listing file prodotto dall'assembler. In sostanza, attraverso queste direttive possiamo dire all'assembler quali porzioni del nostro programma devono essere assemblate e quali no! In effetti le direttive appena illustrate vengono utilizzate principalmente per questo scopo; in questi casi si parla anche di **assemblaggio condizionale**.
Come si puo' notare, questo tipo di blocco condizionale deve iniziare con un **IF** e deve terminare con un **ENDIF**; questa caratteristica rende possibile la creazione di blocchi condizionali innestati. Possiamo scrivere ad esempio:

```
IF (expr1 GT expr2)      ; inizio blocco 1
   IF (expr1 GT expr3)   ; inizio blocco 2
      istruzione1
   ELSE
      istruzione2
   ENDIF                 ; fine blocco2
ELSEIF (expr1 LT expr2)
   istruzione3
ELSE
   istruzione4
ENDIF                    ; fine blocco1
```

All'interno di un blocco **IF ENDIF** le direttive **ELSEIF** e **ELSE** sono facoltative; se necessario si puo' inserire un numero arbitrario di **ELSEIF**, mentre la direttiva **ELSE** deve essere unica in quanto rappresenta la scelta predefinita nel caso in cui tutte le precedenti espressioni siano risultate **FALSE**. L'assembler valuta solo le istruzioni associate alla prima espressione che vale **TRUE**, dopo di che salta alla direttiva **ENDIF** (cioe' esce dal blocco condizionale); anche se piu' espressioni valgono **TRUE**, viene presa in considerazione solo la prima di esse. In assenza di espressioni che valgono **TRUE** vengono valutate le istruzioni associate al caso **ELSE**; se **ELSE** non e' presente l'assembler salta direttamente alla direttiva **ENDIF**.
Tradizionalmente (anche per motivi di chiarezza) le direttive vengono scritte sempre in maiuscolo; in ogni caso e' possibile anche l'uso delle lettere minuscole. Come e' stato gia' detto, possiamo creare espressioni di qualunque complessita'; possiamo scrivere ad esempio:

`IF (expr1 GT (expr2 AND (NOT expr3)));`

se l'espressione e' verificata (**TRUE**) l'assembler effettua l'assemblaggio di tutte le istruzioni che seguono la direttiva **IF**; in caso contrario vengono esaminate le espressioni associate ad eventuali direttive **ESLEIF** e **ELSE**.

Naturalmente e' anche possibile scrivere:

```
IF expr1;
```

questo costrutto significa: se **expr1** e' diversa da zero, cioe' se il risultato prodotto dalla valutazione di **expr1** e' diverso da zero. Attenzione invece al fatto che il costrutto:

```
IF (NOT expr1)
```

non significa: se **expr1** vale zero; infatti l'operatore **NOT** inverte i bit di **expr1** per cui questo costrutto significa: se **NOT expr1** e' diverso da zero. La forma corretta da utilizzare e':

```
IF (expr1 EQ 0).
```

Passiamo ora ad un'altro importante gruppo di assembler time directives che ci permette di prendere delle decisioni in base al fatto che un determinato operando sia stato definito o meno; un tipico blocco condizionale di questo genere comprende le direttive: **IFDEF**, **ELSEIFDEF**, **ELSE**, **ENDIF**. Possiamo scrivere ad esempio:

```
IFDEF OPERANDO1         ; se esiste OPERANDO1
   blocco istruzioni 1
ELSEIFDEF OPERANDO2     ; se esiste OPERANDO2
   blocco istruzioni 2
ELSE                    ; scelta predefinita
   blocco istruzioni 3
ENDIF
```

Il costrutto:

```
IFDEF OPERANDO1
```

significa: se **OPERANDO1** esiste, cioe' se **OPERANDO1** e' stato definito da qualche parte del programma; in caso affermativo viene assemblato il primo blocco di istruzioni. Altrimenti, se esiste **OPERANDO2** viene assemblato il secondo blocco di istruzioni; se nessuno dei casi precedenti e' verificato, viene presa la decisione predefinita che consiste nell'assemblaggio del terzo blocco di istruzioni. In queste valutazioni non ha nessuna importanza il valore assegnato ad un operando; infatti anche se scriviamo:

```
OPERANDO1 = 0,
```

il test:

```
IFDEF OPERANDO1
```

restituisce **TRUE** in quanto stiamo testando se **OPERANDO1** esiste e non se e' diverso da zero.

Vediamo un esempio pratico che dimostra la grande utilita' di queste direttive; per determinare la dimensione in byte di un tipo di dato, il **TASM** mette a disposizione l'operatore **SIZE**. Se ad esempio il nome simbolico **Abbonato1** rappresenta un'istanza di una struttura grande **24** byte, allora l'istruzione:

```
mov ax, SIZE Abbonato1
```

carica in **AX** il valore **24**, che rappresenta appunto lo spazio in byte occupato in memoria da **Abbonato1**. Per ottenere questa stessa informazione, il **MASM** utilizza invece l'operatore **SIZEOF**; se vogliamo fare in modo che lo stesso codice sorgente venga assemblato correttamente sia da **MASM** che da **TASM** possiamo scrivere:

```
TASM = 1;

IFDEF TASM
   mov      ax, SIZE Abbonato1
ELSE
   mov      ax, SIZEOF Abbbonato1
ENDIF
```

Se vogliamo assemblare il programma con il **TASM** possiamo dichiarare il nome simbolico **TASM** che provochera' l'assemblaggio solo della prima istruzione; infatti in questo caso il costrutto **IFDEF TASM** restituisce **TRUE** perche' il nome **TASM** esiste. Se invece vogliamo assemblare il programma con il **MASM** ci basta eliminare (o commentare) la dichiarazione del nome simbolico **TASM**; in questo caso verra' assemblata solo la seconda istruzione.
Alternativamente e' anche possibile scrivere:

```
TASM = 0;

IF TASM
   mov      ax, SIZE Abbonato1
ELSE
   mov      ax, SIZEOF Abbbonato1
ENDIF
```

In questo modo, assegnando al nome simbolico **TASM** un valore diverso da zero, otteniamo l'assemblaggio solo della prima istruzione; assegnando invece a **TASM** il valore zero, otteniamo l'assemblaggio solo della seconda istruzione.
Sono disponibili anche le direttive **IFNDEF**, **ELSEIFNDEF** che seguono una logica inversa rispetto a **IFDEF**, **ELSEIFDEF**; ad esempio, il costrutto:
`IFNDEF OPERANDO1,`
significa: se **OPERANDO1** non e' stato definito (not defined). Possiamo dire quindi che se **OPERANDO1** non esiste, il costrutto precedente restituisce **TRUE**; se invece **OPERANDO1** esiste, si ottiene **FALSE**.
Piu' avanti verranno illustrate una serie di altre importanti assembler time directives che vengono prevalentemente usate per creare le macro.

Le direttive condizionali per la fase di esecuzione.

Come e' stato accennato in precedenza, le versioni piu' recenti di **TASM** (**5.x** o superiore) e **MASM** (**6.x** o superiore) forniscono una serie di direttive e operatori utilizzabili anche a run time; attraverso questi strumenti e' possibile scrivere programmi Assembly utilizzando una sintassi molto simile a quella dei linguaggi di alto livello come il **C** e il **Pascal**.
Tutte le run time directives devono essere precedute da un punto (**.**) che le distingue dalle assembler time directives; partiamo dalle direttive condizionali che sono: **.IF**, **.ELSEIF**, **.ELSE** e **.ENDIF**. I run time operators possono essere utilizzati esclusivamente in combinazione con le run time directives; possiamo scrivere ad esempio:

```
.IF (ax < bx)           ; se ax e' minore di bx
   mov      dx, 10
   add      Var1, dx
.ELSEIF (ax == bx)      ; se ax e' uguale a bx
   mov      dx, 20
   add      Var2, dx
.ELSE                   ; se ax e' maggiore di bx
   mov      dx, 30
   add      Var3, dx
.ENDIF
```

Un programmatore Assembly con un minimo di esperienza non si fa certo impressionare da questa "meraviglia" e intuisce subito il trucco; per svelare il mistero ci basta consultare il listing file del nostro programma; nel caso ad esempio del **TASM**, scopriamo che il codice precedente viene tradotto (com'era prevedibile) nel seguente modo:

```
   cmp      ax, bx          ; compara ax con bx
   jae      @C0001          ; se (ax >= bx) salta a @C0001
   mov      dx, 10          ; altrimenti assembla queste
   add      Var1, dx        ; due istruzioni
   jmp      @C0000          ; e salta a .ENDIF
@C0001:                     ; etichetta (ax >= bx)
   cmp      ax, bx          ; compara ax con bx
   jnz      @C0002          ; se sono diversi salta a @C0002
   mov      dx, 20          ; altrimenti assembla queste
   add      Var2, dx        ; due istruzioni
   jmp      @C0000          ; e salta a .ENDIF
@C0002:                     ; etichetta (ax >= bx)
   mov      dx, 30          ; assembla queste
   add      Var3, dx        ; due istruzioni
@C0000:                     ; .ENDIF
```

Come si puo' notare, le run time directives non hanno niente di magico; queste direttive infatti vengono interpretate dall'assembler che le traduce poi nella corrispondente sequenza di istruzioni Assembly.
Le run time directives presentano una importante limitazione rappresentata dal fatto che l'espressione booleana associata ad esse deve essere costituita da un confronto tra due operandi semplici; non e' possibile quindi utilizzare espressioni complesse del tipo:
`.IF (ax < (bx & !cx))`.
Nel caso generale, una direttiva condizionale viene tradotta in un confronto tra l'operando di sinistra e quello di destra; ne consegue che questi due operandi debbono soddisfare i requisiti richiesti da istruzioni come **CMP**, **TEST** e da molte altre istruzioni dell'Assembly. In sostanza, anche in questo caso possiamo parlare di operando sorgente e operando destinazione; sono consentiti quindi confronti del tipo:

destinazione	sorgente
registro	registro
registro	memoria
registro	immediato
memoria	registro
memoria	immediato

Sono proibiti invece i confronti del tipo:

destinazione	sorgente
memoria	memoria
immediato	memoria
immediato	registro

Oltre alle direttive condizionali sono disponibili anche una serie di direttive che consentono di implementare le iterazioni; anche in questo caso si utilizza la classica sintassi delle analoghe direttive offerte dai linguaggi di alto livello. Il ciclo **while do** viene implementato con le direttive **.WHILE**, **.ENDW**; il ciclo **repeat until** viene implementato con le direttive **.REPEAT**, **.UNTIL** e **.UNTILCXZ**.
Supponiamo di voler inizializzare con il valore **3BF2h** tutti i **10** elementi di un vettore chiamato **Vector1**; attraverso un ciclo **while do** possiamo scrivere:

```
MAX_LOOPS = 10                  ; numero elementi

   mov      bx, offset Vector1  ; vettore da inizializzare
   mov      ax, 3BF2h           ; valore inizializzante
   mov      cx, 0               ; contatore
.WHILE (cx < MAX_LOOPS)         ; condizione di iterazione
   mov      [bx], ax            ; inizializza l'elemento
   add      bx, 2               ; incremento puntatore
   inc      cx                  ; incremento contatore
.ENDW                           ; salta a .WHILE
```

Come accade con i linguaggi di alto livello, la direttiva **.WHILE** valuta l'espressione tra parentesi tonde, e se ottiene **TRUE** esegue le istruzioni successive; dopo l'ultima istruzione, la direttiva **.ENDW** salta nuovamente a **.WHILE** per cui il test viene nuovamente ripetuto. Le iterazioni continuano purche' l'espressione tra parentesi tonde continui ad essere **TRUE**; quando l'espressione diventa **FALSE** l'iterazione termina e l'esecuzione riprende dall'istruzione successiva alla direttiva **.ENDW**. Appare evidente che se l'espressione booleana risulta **FALSE** sin dall'inizio, il ciclo **.WHILE** non viene mai eseguito; inoltre, se l'espressione booleana non diventa mai **FALSE**, si innesca un loop infinito.
Nel caso del **TASM** il codice precedente viene tradotto nel seguente modo:

```
MAX_LOOPS = 10                  ; numero elementi

   mov      bx, offset Vector1  ; vettore da inizializzare
   mov      ax, 3BF2h           ; valore inizializzante
   mov      cx, 0               ; contatore
@C0000:                         ; etichetta inizio loop
   cmp      cx, MAX_LOOPS       ; compara cx con MAX_LOOPS
   jae      @C0001              ; se (cx >= MAX_LOOPS) salta a @C0001
   mov      [bx], ax            ; inizializza l'elemento
   add      bx, 2               ; incremento puntatore
   inc      cx                  ; incremento contatore
   jmp      @C0000              ; ricomincia
@C0001:                         ; uscita dal loop
```

Se abbiamo bisogno di un ciclo che venga eseguito almeno una volta possiamo utilizzare le direttive **.REPEAT** e **.UNTIL**; in questo caso infatti la condizione di uscita dal ciclo viene testata alla fine del ciclo stesso. La struttura generale e' la seguente:

```
.REPEAT
   sequenza istruzioni
.UNTIL (espressione)
```

Come si puo' notare, la **sequenza istruzioni** viene eseguita almeno una volta; alla fine viene valutata l'**espressione** associata alla direttiva **.UNTIL**. Se **espressione** risulta **FALSE** viene ripetuto il ciclo; le iterazioni continuano finche' **espressione** non diventa **TRUE**.
Al posto di **.UNTIL** si puo' utilizzare anche la direttiva **.UNTILCXZ**; come si puo' facilmente

intuire, questa direttiva termina le iterazioni quando **espressione** diventa **TRUE** e/o quando **CX** diventa zero.

Per creare dei loop piu' sofisticati sono disponibili anche le direttive **.BREAK** e **.CONTINUE**; queste due direttive sono analoghe a quelle fornite dai linguaggi di alto livello. All'interno di un loop, la direttiva **.BREAK** provoca l'immediata uscita dal loop stesso; si puo' scrivere ad esempio:

```
.BREAK .IF (ax == cx).
```

Sempre all'interno di un loop, la direttiva **.CONTINUE** determina un salto alla direttiva **.WHILE** o **.UNTIL** o **UNTILCXZ**, provocando quindi una nuova valutazione della condizione di uscita; si puo' scrivere ad esempio:

```
.CONTINUE .IF (ax < cx).
```

In definitiva, il comportamento delle run time directives e' assolutamente analogo a quello delle istruzioni per il controllo di flusso dei linguaggi di alto livello; pertanto, se si vogliono avere maggiori dettagli sull'uso di tutte queste direttive si consiglia di consultare un qualsiasi manuale di programmazione sul linguaggio **C**, **Pascal**, etc.

Non ci vuole molto a capire che le run time directives fanno storcere il naso a parecchi programmatori Assembly; in effetti, non si capisce bene che senso abbia programmare in questo modo. L'aspetto piu' discutibile poi e' dato dal fatto che queste direttive vengono tradotte in istruzioni Assembly in base ai gusti di chi ha progettato l'assembler; e non e' detto che questi gusti coincidano con quelli dei programmatori. Con le run time directives stiamo dicendo all'assembler di scrivere il codice al nostro posto; tutto cio' e' chiaramente in contrasto con la filosofia di quei programmatori che utilizzano l'Assembly proprio perche' vogliono avere il controllo totale sul programma che stanno scrivendo. Se ritenete che le run time directives siano interessanti allora l'Assembly non fa al caso vostro; tanto vale passare direttamente ad un linguaggio di alto livello che vi mette a disposizione tutte queste comodita'. Una valida alternativa e' rappresentata dal Linguaggio **C** che puo' essere considerato un vero e proprio Assembly di alto livello.

Le macro.

Molti linguaggi di programmazione, compreso l'Assembly, supportano le **macro**; si tratta di un potentissimo strumento che permette di rappresentare con un nome simbolico, intere porzioni di codice, se non addirittura interi programmi. Si puo' tracciare un parallelo tra la struttura di una macro e la definizione di una variabile; la definizione di una variabile consiste in un nome simbolico, una direttiva (**DB**, **DW**, etc) che definisce il tipo di variabile e il valore numerico di inizializzazione. Nel caso della macro, la differenza sostanziale sta nel fatto che al posto del valore numerico di inizializzazione ci puo' essere praticamente qualsiasi cosa; una macro infatti puo' rappresentare un semplice valore numerico o anche una sequenza di caratteri alfanumerici che nel loro insieme possono formare intere porzioni di codice. In fase di assemblaggio di un programma, ogni volta che l'assembler incontra il nome di una macro, lo sostituisce con tutto cio' che la macro stessa rappresenta.
Un'altra differenza sostanziale rispetto alle variabili e' data dal fatto che una macro e' una semplice dichiarazione a disposizione dell'assembler; come tutte le dichiarazioni quindi, la macro non ha un indirizzo di memoria. Questo significa che la dichiarazione di una macro puo' essere collocata dappertutto, sia all'interno, sia all'esterno dei segmenti di programma.

Il caso piu' semplice in assoluto e' rappresentato da un operando immediato dichiarato attraverso la direttiva di assegnamento =; possiamo creare quindi una semplicissima macro scrivendo ad esempio:

```
PESO_KG = 300.
```

In fase di assemblaggio del programma, ogni volta che l'assembler incontra il nome **PESO_KG** lo

sostituisce con il valore numerico **300**; si consiglia di utilizzare sempre nomi simbolici al posto dei numeri espliciti. In questo modo si rende il programma piu' comprensibile e si riduce notevolmente il rischio di commettere errori; se si sbaglia nello scrivere un numero, l'assembler non e' in grado di intervenire, mentre se si sbaglia nello scrivere un nome simbolico, si ottiene subito un messaggio di errore da parte dell'assembler.

Con l'operatore di assegnamento = possiamo dichiarare nomi simbolici che rappresentano esclusivamente valori numerici; se ci serve qualcosa di piu' sofisticato dobbiamo ricorrere alla direttiva **EQU** (equivalent). Con questa direttiva possiamo dichiarare nomi simbolici che possono rappresentare sia valori numerici che stringhe alfanumeriche; l'unica limitazione e' data dal fatto che una dichiarazione fatta con la direttiva **EQU** deve svilupparsi su un'unica linea del programma (non e' consentito cioe' andare a capo). Con la direttiva **EQU** la dichiarazione precedente puo' essere riscritta come:

```
PESO_KG EQU 300.
```

A differenza di quanto accade con la direttiva =, se abbiamo dichiarato **PESO_KG** con la direttiva **EQU** non e' possibile ridichiarare questo nome simbolico in nessun'altra parte del programma; non e' possibile cioe' assegnare a **PESO_KG** un'altro valore o un'altro significato. Questa limitazione riguarda solo i nomi simbolici dichiarati con **EQU** che rappresentano valori numerici; la ridichiarazione invece e' consentita per i nomi simbolici dichiarati con **EQU** che rappresentano stringhe alfanumeriche.
Una macro dichiarata con **EQU** e' formata da un nome simbolico, dalla direttiva **EQU** e dal cosiddetto **corpo** della macro; in fase di assemblaggio di un programma, l'assembler ogni volta che incontra il nome di una macro, lo sostituisce con il corpo della macro stessa. Questo procedimento si chiama **espansione della macro**; il corpo di una macro puo' rappresentare qualsiasi cosa che abbia un senso logico nel contesto del programma che si sta scrivendo. In sostanza, dopo la fase di espansione delle macro si deve ottenere codice Assembly sensato; in caso contrario l'assembler genera gli opportuni messaggi di errore. Supponiamo di voler rappresentare una stringa **C** mediante una macro; possiamo scrivere:

```
TEXT_MACRO EQU Stringa da stampare, 0.
```

A questo punto nel blocco dati del nostro programma possiamo scrivere:

```
Stringa1 db TEXT_MACRO.
```

In fase di assemblaggio del programma, l'assembler esaminando il blocco dati incontra il nome simbolico **TEXT_MACRO** e lo sostituisce con il suo corpo ottenendo:

```
Stringa1 db Stringa da stampare, 0.
```

Chiaramente questo codice e' privo di senso per cui l'assembler genera un messaggio di errore; ovviamente il modo corretto di scrivere la macro e':

```
TEXT_MACRO EQU 'Stringa da stampare', 0.
```

Una volta capito il meccanismo possiamo anche scrivere ad esempio:

```
MACRO_DB EQU db 'Stringa da stampare', 0,
```

o direttamente:

```
MACRO_STRING_DB EQU Stringa1 db 'Stringa da stampare', 0.
```

Utilizzando quest'ultima macro, nel blocco dati del nostro programma possiamo scrivere semplicemente:

```
MACRO_STRING_DB.
```

Quando l'assembler incontra questo nome simbolico lo espande in:

```
Stringa1 db 'Stringa da stampare', 0.
```

Se, subito dopo la dichiarazione della macro **MACRO_STRING_DB**, scriviamo:

```
MACRO_2 EQU MACRO_STRING_DB,
```

l'assembler si accorge che il nome simbolico **MACRO_STRING_DB** appartiene ad una macro gia' dichiarata in precedenza, e quindi assegna il corpo di **MACRO_STRING_DB** a **MACRO_2**; in

sostanza, dopo queste dichiarazioni, anche il corpo di **MACRO_2** vale:

```
Stringa1 db 'Stringa da stampare', 0.
```

Nota: Il **Borland TASM** consiglia di racchiudere sempre tra parentesi angolari il corpo di una macro dichiarata con **EQU**.

La direttiva **EQU** si rivela utilissima in molte circostanze; in particolare, la possiamo utilizzare per implementare nuove istruzioni che non sono supportate dall'assembler che stiamo usando.
Supponiamo ad esempio di avere un assembler che non supporta l'istruzione **CPUID**; come gia' sappiamo, il codice macchina di questa istruzione e' formato dai due opcodes **00001111b = 0Fh** e **10100010b = A2h**. Con l'ausilio di **EQU** ci basta scrivere:

```
CPUID EQU db 00001111b, 10100010b,
```

oppure:

```
CPUID EQU db 0Fh, 0A2h;
```

in questo modo, anche il nostro vecchio assembler sara' in grado di supportare l'istruzione **CPUID**.

Se nemmeno la direttiva **EQU** soddisfa le nostre esigenze, allora non ci resta che ricorrere alle **multiline macro** (macro multilinea); con le macro multilinea si puo' fare veramente di tutto ed e' difficile che un programmatore possa pretendere di piu'.
La struttura generale di una macro multilinea e' formata da un nome simbolico, dalla direttiva **MACRO**, da una lista facoltativa di parametri e dal corpo della macro; al termine del corpo della macro deve essere presente la direttiva **ENDM** (end macro). La differenza sostanziale rispetto a **EQU** sta nel fatto che le macro dichiarate con la direttiva **MACRO** possono svilupparsi eventualmente su piu' linee; inoltre, come e' stato appena detto, una macro multilinea puo' essere dotata anche di una lista di parametri. Vediamo alcuni esempi pratici che illustrano le potenzialita' di questo strumento.
Nei precedenti capitoli abbiamo visto che con le CPU della famiglia **80x86** non e' consentito il trasferimento dati da memoria a memoria; possiamo sopperire a questa carenza attraverso la seguente macro multilinea:

```
MOV_MEM  MACRO Destinazione, Sorgente

   push     Sorgente
   pop      Destinazione

ENDM
```

Come si puo' notare, questa macro e' formata da un nome **MOV_MEM** che simboleggia una istruzione **MOV** tra due generici operandi; abbiamo poi la direttiva **MACRO** seguita da una lista formata dai due parametri **Destinazione** e **Sorgente** separati da virgole. Il corpo della macro e' formato da un'istruzione **PUSH** che inserisce nello stack il contenuto di **Sorgente**, e da un'istruzione **POP** che estrae dallo stack questo contenuto e lo salva in **Destinazione** realizzando cosi' il trasferimento dati richiesto. Infine, al termine del corpo della macro e' presente la direttiva **ENDM**.
A questo punto all'interno del blocco codice del nostro programma possiamo scrivere istruzioni del tipo:

```
MOV_MEM Var1, Var2
```

(dove **Var1** e **Var2** sono ad esempio due variabili a **16** bit). In fase di assemblaggio del programma, quando l'assembler incontra la precedente istruzione, effettua l'espansione della macro **MOV_MEM**; questa espansione consiste anche nel sostituire al parametro **Destinazione** l'argomento **Var1**, e al parametro **Sorgente** l'argomento **Var2**.

Alla fine, come risulta anche dal listing file, si ottiene:

```
push     Var2
pop      Var1
```

In questo modo abbiamo l'impressione di copiare **Var2** in **Var1** con un'unica istruzione; come si puo' notare, la macro **MOV_MEM** senza nessuna modifica gestisce anche operandi a **32** bit. Come al solito, e' fondamentale il fatto che l'espansione della macro porti alla generazione di codice che abbia un senso logico; se ad esempio **Var2** e' un operando di tipo **mem8**, allora l'istruzione:
`MOV_MEM Var1, Var2`
provoca un messaggio di errore dell'assembler in quanto stiamo tentando di inserire nello stack un operando a **8** bit. Bisogna anche prestare molta attenzione al fatto che l'uso disinvolto delle macro puo' portare a degli errori piuttosto subdoli e quindi difficili da individuare; nel caso ad esempio della macro **MOV_MEM**, nessuno ci impedisce di scrivere:
`MOV_MEM ax, bx.`
In questo caso l'assembler utilizza **BX** come sorgente e **AX** come destinazione, generando codice Assembly del tutto corretto; se pero' scriviamo:
`MOV_MEM ax, ebx,`
otteniamo un'istruzione **PUSH** che inserisce nello stack i **32** bit di **EBX**, e un'istruzione **POP** che estrae dallo stack **16** bit e li copia in **AX**. L'assembler non rileva ovviamente nessun errore in quanto il codice appena espanso e' perfettamente legale; alla fine ci ritroviamo con **16** bit di dati in eccesso nello stack (stack disallineato).

Come e' stato detto in precedenza, il corpo di una macro multilinea puo' contenere praticamente qualsiasi cosa che abbia un senso logico; vediamo un'altro esempio di una macro chiamata **INIT_VECTOR** che inizializza con il valore **InitVal** i **NumElem** elementi da **NumByte** byte ciascuno di un generico vettore che si trova all'offset **VectorOffset**:

```
INIT_VECTOR MACRO InitVal, NumElem, VectorOffset, NumByte

   mov      cx, NumElem
init_loop:
   mov      [VectorOffset], InitVal
   add      VectorOffset, NumByte
   loop     init_loop

ENDM
```

Come si puo' notare, si tratta del classico loop gestito da **CX** per inizializzare gli elementi di un vettore; supponiamo ora di voler inizializzare con il valore **3500** i **10** elementi da **2** byte ciascuno di un vettore **Vector1**. Possiamo scrivere allora:

```
NUM_ELEMENTI = 10
VALORE_INIZIALE = 3500

   mov      bx, offset Vector1
   mov      ax, VALORE_INIZIALE
   INIT_VECTOR ax, NUM_ELEMENTI, bx, 2
```

L'espansione della macro portera' alla generazione del seguente codice:

```
   mov      cx, NUM_ELEMENTI
init_loop:
   mov      [bx], ax
   add      bx, 2
   loop     init_loop
```

E' chiaro che il programmatore e' tenuto a chiamare la macro passandole il corretto numero di argomenti; questi argomenti inoltre debbono avere determinate caratteristiche in modo che l'espansione della macro porti alla generazione di codice legale. In caso contrario (se si e' fortunati), si ottengono dei messaggi di errore da parte dell'assembler; non e' possibile ad esempio passare alla macro il valore immediato **VALORE_INIZIALE** come argomento **InitVal** in quanto si otterrebbe un'istruzione:

```
mov [bx], VALORE_INIZIALE
```

che non specifica la dimensione in bit del trasferimento dati.

La struttura della macro **INIT_VECTOR** ci permette di analizzare un problema che si verifica quando, nel corpo di una macro e' presente un'etichetta (nel nostro caso **init_loop**); se ad esempio, nel nostro programma sono presenti due chiamate alla macro **INIT_VECTOR**, verranno effettuate due espansioni che determineranno la presenza di due etichette (**init_loop**) aventi lo stesso identico nome. Naturalmente questa e' una situazione illegale in quanto e' proibito ridefinire un nome gia' definito in precedenza; una possibile soluzione a questo problema e' rappresentata dall'uso della direttiva **LOCAL**. La direttiva **LOCAL** se presente, deve essere la prima istruzione del corpo della macro; questa direttiva elenca una serie di nomi simbolici che verranno espansi dall'assembler sempre con nomi diversi in modo da evitare le ridefinizioni. Nel caso quindi della macro **INIT_VECTOR** possiamo scrivere:

```
INIT_VECTOR MACRO InitVal, NumElem, VectorOffset, NumByte
   LOCAL    init_loop

   mov      cx, NumElem
init_loop:
   mov      [VectorOffset], InitVal
   add      VectorOffset, NumByte
   loop     init_loop

ENDM
```

Se questa macro viene ad esempio espansa 10 volte nel blocco codice del nostro programma, otterremo 10 nomi differenti per l'etichetta **init_loop**; questi nomi vengono scelti direttamente dall'assembler. Si tenga presente che nel caso del **TASM** non e' consentito lasciare righe vuote tra l'intestazione della macro e la direttiva **LOCAL**; in caso contrario si ottengono strani messaggi di errore da parte dell'assembler.

Tornando al caso generale, bisogna dire che il corpo di una macro multilinea puo' contenere chiamate ad altre macro che naturalmente devono essere gia' state dichiarate in precedenza; e' anche consentito l'utilizzo di tutte le assembler time directives e degli assembler time operators. In questo modo e' possibile indicare all'assembler quali parti di una macro devono essere espanse e quali no; e' consigliabile comunque non abusare troppo di queste caratteristiche per evitare di scrivere programmi troppo contorti.
E' addirittura possibile dichiarare macro ricorsive; una macro e' ricorsiva quando presenta nel suo corpo una chiamata a se stessa. Le macro ricorsive verranno illustrate in un apposito capitolo

dedicato all'implementazione della ricorsione in Assembly.

Un'aspetto molto importante da chiarire riguarda il fatto che qualcuno, osservando la struttura e le caratteristiche di una macro multilinea, potrebbe pensare di trovarsi davanti ad un vero e proprio sottoprogramma; in effetti l'analogia esiste perche', come vedremo nel prossimo capitolo, un sottoprogramma e' formato da un nome, da una direttiva e da un corpo. Questa analogia pero' e' solamente formale perche' in realta' tra le macro e i sottoprogrammi esiste una notevole differenza sostanziale; come accade per le normali variabili, anche un sottoprogramma occupa memoria e quindi ha anche un indirizzo che rappresenta l'indirizzo da cui inizia il blocco di memoria che contiene il corpo del sottoprogramma stesso. Come e' stato detto in precedenza, una macro rappresenta invece una semplice dichiarazione attraverso la quale diamo all'assembler una serie di indicazioni; nella fase di assemblaggio l'assembler utilizza queste indicazioni per effettuare l'espansione della macro.
Un'altra questione molto importante riguarda il fatto che su alcuni libri si afferma che le macro multilinea in determinate circostanze possono rappresentare una valida alternativa ai sottoprogrammi; questa affermazione e' corretta purche' si tenga sempre presente la differenza sostanziale che esiste tra macro e sottoprogrammi, cioe' tra dichiarazioni e definizioni. Come vedremo nel prossimo capitolo, un sottoprogramma viene caricato in memoria assieme al programma principale; il programma principale puo' chiamare un sottoprogramma quando vuole e dove vuole. Ciascuna di queste chiamate corrisponde ad un salto (trasferimento del controllo) all'indirizzo di memoria da cui inizia il corpo del sottoprogramma; al termine del sottoprogramma, il controllo torna al programma principale (o in generale, a chi ha effettuato la chiamata). Questa tecnica di programmazione presenta il vantaggio di farci risparmiare parecchi byte di codice; infatti, se il programma principale deve chiamare 100 volte un determinato sottoprogramma, non vengono effettuati 100 salti a 100 copie diverse del sottoprogramma, ma 100 salti sempre allo stesso sottoprogramma. Lo svantaggio evidente di questa tecnica di programmazione e' rappresentato invece dal sensibile calo delle prestazioni; e' chiaro infatti che i continui salti dal programma principale ai sottoprogrammi e viceversa, determinano una certa diminuzione della velocita' di esecuzione.
Se al posto di un sottoprogramma si utilizza una macro, si ottengono vantaggi e svantaggi di segno opposto; se il programma principale deve chiamare 100 volte una stessa macro, si ottengono 100 espansioni del corpo della macro, che fanno aumentare notevolmente le dimensioni del codice. Il vantaggio delle macro sta nel fatto che il loro uso elimina parecchi salti favorendo quindi uno stile di programmazione di tipo sequenziale; tanto minore e' il numero di salti, tanto maggiore sara' la velocita' del programma.
In definitiva, se il nostro programma ha bisogno ad esempio di utilizzare piu' volte uno stesso algoritmo, possiamo inserire questo algoritmo in un sottoprogramma o in una macro; per decidere quale sia la scelta migliore basta fare un semplice ragionamento. Se l'algoritmo e' molto grosso e viene chiamato parecchie volte dal programma principale, conviene inserirlo in un sottoprogramma; in questo modo si ottiene un programma molto compatto e sufficientemente veloce. Se l'algoritmo e' relativamente piccolo e viene chiamato poche volte dal programma principale, conviene inserirlo in una macro multilinea; in questo modo si ottiene un programma sufficientemente compatto e molto veloce.

Le potenzialita' delle macro multilinea vanno ben oltre gli aspetti appena illustrati; per descrivere in modo esauriente questo argomento non basterebbe un intero libro. Per maggiori dettagli si puo' fare riferimento ai manuali del proprio assembler; si tenga presente comunque che certe caratteristiche delle macro possono variare sintatticamente da assembler ad assembler creando quindi problemi di compatibilita'.

Le macro predefinite di MASM e TASM.

Nel precedente paragrafo sono stati analizzati gli strumenti con i quali si possono creare macro personalizzate; il **MASM** e il **TASM** dispongono anche di altri strumenti con i quali si possono creare macro aventi una struttura predefinita. Esaminiamo in particolare le macro che si possono creare con le direttive **REPT** e **WHILE**.
La direttiva **REPT** (repeat) ci permette di creare macro il cui corpo viene ripetuto un certo numero di volte; il numero di ripetizioni da effettuare viene indicato da un valore immediato che deve essere specificato subito dopo la direttiva **REPT**. La struttura generale di questa macro predefinita e' la seguente:

```
REPT contatore
   corpo della macro
ENDM
```

Utilizziamo ad esempio **REPT** per sopperire ad una carenza delle CPU **8086** che non supportano istruzioni del tipo:
`shl ax, 4.`
Invece di inserire 4 istruzioni ciascuna delle quali fa scorrere i bit di **AX** di una posizione verso sinistra, possiamo scrivere la seguente macro:

```
REPT 4
   shl   ax, 1
ENDM
```

Questa macro deve essere inserita direttamente nel blocco codice del programma; subito dopo la fase di assemblaggio, la macro viene convertita nelle seguenti quattro istruzioni:

```
shl      ax, 1
shl      ax, 1
shl      ax, 1
shl      ax, 1
```

La direttiva **WHILE** ci permette di creare macro il cui corpo viene ripetuto un numero di volte che dipende dal risultato della valutazione di un'espressione booleana; l'espressione booleana deve essere specificata subito dopo la direttiva **WHILE**. La struttura generale di questa macro predefinita e' la seguente:

```
WHILE espressione
   corpo della macro
ENDM
```

In pratica, viene valutata **espressione**, e se si ottiene **TRUE** viene espanso il corpo della macro; successivamente viene nuovamente valutata **espressione**, e se si ottiene ancora **TRUE** si verifica una nuova espansione del corpo della macro. Il processo continua finche' **espressione** non diventa **FALSE**; in casi di questo genere e' importante evitare come al solito di creare un loop infinito.
Molto spesso, la direttiva **WHILE** viene usata in combinazione con la direttiva **EXITM** (exit macro) che permette di uscire immediatamente da una qualsiasi macro multilinea; la sintassi per l'utilizzo di **EXITM** e' la seguente:

```
IF espressione
   EXITM
ENDIF
```

Capitolo 25 - I sottoprogrammi

Moltissimi linguaggi di programmazione, compreso l'Assembly, offrono il pieno supporto per i **sottoprogrammi**; il sottoprogramma e' un vero e proprio "mini-programma" che consente di isolare una porzione di codice dal programma principale. In questo senso il sottoprogramma puo' essere visto come una sorta di **modulo** che esegue un compito specifico; ogni volta che il nostro programma deve eseguire quel particolare compito, non deve fare altro che "chiamare" il relativo sottoprogramma. L'uso dei sottoprogrammi presenta diversi aspetti di notevole interesse; analizziamo in particolare gli aspetti piu' importanti:

Programmazione modulare. Lo stile di programmazione che consiste nel suddividere un programma in tanti moduli, cioe' in tanti sottoprogrammi, prende il nome di **programmazione modulare**; in questo caso l'esecuzione di un programma consiste in una serie di chiamate ai vari sottoprogrammi che lo compongono.

Ricerca semplificata degli errori. In un programma modulare la ricerca degli eventuali errori viene notevolmente semplificata; infatti, un programma modulare che si comporta in modo non corretto, permette al programmatore di intuire facilmente quale puo' essere il modulo responsabile dell'anomalia. Nel caso di un programma tradizionale il programmatore e' costretto invece a riesaminare tutto il codice sorgente.

Riusabilita' del codice. Nella programmazione modulare, capita spesso di scrivere moduli particolarmente interessanti ed efficienti che possono tornare utili anche in altri programmi; in un caso del genere e' facilissimo prelevare da un programma il modulo che ci interessa, per inserirlo poi in un'altro programma. Una diretta conseguenza di questo aspetto e' rappresentata dalla possibilita' di raccogliere in un unico file una serie di moduli aventi caratteristiche comuni; questi particolari files prendono il nome di **librerie di sottoprogrammi**. Pensiamo ad esempio ad una libreria di moduli per la gestione dei files su disco, una libreria di moduli per la gestione della scheda video, etc; queste librerie, una volta realizzate, possono essere facilmente "linkate" ai programmi che ne hanno bisogno.

Facilita' di manutenzione. Un programma modulare puo' essere facilmente modificato o aggiornato senza la necessita' di riscriverlo da zero; infatti, tutto il lavoro di modifica o di aggiornamento si concentra sui singoli moduli e non sull'intero programma. Esiste quindi anche la possibilita' di modificare o aggiornare un solo modulo che non soddisfa le nostre esigenze; nel caso dei programmi tradizionali, le modifiche sono estremamente difficili se non impossibili in quanto c'e' l'elevato rischio di compromettere in modo irrimediabile l'intero programma. A tale proposito e' emblematico il caso di molti vecchi e famosi programmi scritti secondo il classico stile (si fa per dire) sequenziale del **BASIC**; al momento di aggiornare questi programmi per adattarli al nuovo hardware disponibile, ci si e' resi conto che sarebbe stato molto piu' conveniente (in termini di tempo e di soldi) riscriverli da zero.

Nel linguaggio **C** i sottoprogrammi vengono chiamati **funzioni**; il **Pascal** mette a disposizione due tipi di sottoprogrammi attraverso le direttive **function** e **procedure**. Anche in **FORTRAN** vengono supportati due tipi di sottoprogrammi attraverso le direttive **FUNCTION** e **SUBROUTINE**; le **subroutines** sono note anche ai programmatori **BASIC**.
In Assembly i sottoprogrammi vengono chiamati **procedure**; i linguaggi di alto livello sfruttano proprio le procedure dell'Assembly per implementare i sottoprogrammi. Si possono notare delle analogie tra la struttura di un sottoprogramma e la struttura di una funzione matematica; una funzione matematica e' costituita da un **nome**, una **lista di parametri**, un **corpo** e un **valore di ritorno**. Il nome ci permette di identificare la funzione in modo univoco; la lista di parametri

rappresenta l'insieme delle informazioni di cui la funzione ha bisogno per svolgere il proprio lavoro. Il corpo della funzione contiene l'algoritmo che elabora i parametri in ingresso; il valore di ritorno rappresenta il risultato delle elaborazioni compiute dalla funzione. Supponiamo ad esempio di voler scrivere una funzione matematica che calcola l'area di un triangolo; una funzione di questo genere ha bisogno quindi di ricevere in input la base e l'altezza del triangolo. Il corpo della funzione elabora questi parametri e restituisce in output un valore di ritorno che rappresenta l'area del triangolo; indicando con **x** la base del triangolo, con **y** l'altezza, con **z** il valore di ritorno e con **Area** il nome della funzione, possiamo scrivere simbolicamente:

```
z = Area(x, y).
```

Questa simbologia indica il fatto che **z** e' funzione di **x** e di **y**; in sostanza, il valore di **z** dipende dai valori di **x** e di **y**, e per questo motivo si dice anche che **x** e **y** sono **variabili indipendenti**, mentre **z** e' una **variabile dipendente**. La rappresentazione simbolica di **Area** fornisce una descrizione "esterna" delle caratteristiche generali della funzione; nei linguaggi di programmazione questa descrizione e' una dichiarazione che prende il nome di **prototipo di funzione**. Naturalmente, oltre alla dichiarazione della funzione, abbiamo bisogno anche della definizione della funzione, cioe' della creazione materiale del corpo della funzione; nel caso di **Area**, il corpo di questa funzione e' rappresentato ovviamente dal noto algoritmo per il calcolo dell'area di un triangolo:

```
z = (x * y) / 2.
```

Una volta che questo algoritmo e' stato scritto, collaudato e verificato, possiamo anche dimenticarcelo; l'importante e' conoscere il prototipo della funzione che ci indica il procedimento che dobbiamo seguire per "chiamare" la funzione stessa e per ottenere il valore di ritorno. Questo procedimento rappresenta la cosiddetta **interfaccia** della funzione; un programma che ha la necessita' di chiamare una determinata funzione, ne deve conoscere l'interfaccia.
Tornando alla nostra funzione **Area**, supponiamo di voler calcolare l'area di un triangolo avente base **b** e altezza **a**; se vogliamo salvare il risultato in una variabile chiamata **c**, possiamo scrivere:

```
c = Area(b, a).
```

Questa fase rappresenta la **chiamata** della funzione; in questo modo **Area** riceve in input **b** ed **a**, e produce in output un risultato che noi memorizziamo in **c**. I dati **x** e **y** indicati nel prototipo della funzione o nella definizione della funzione prendono il nome di **parametri**, mentre i dati **b** e **a** passati nella chiamata della funzione prendono il nome di **argomenti**; in alcuni testi i parametri vengono chiamati **parametri formali** o **argomenti formali**, mentre gli argomenti vengono chiamati **parametri attuali** o **argomenti reali**.
Non e' detto che una funzione debba per forza restituire un valore di ritorno; esistono funzioni che devono solo eseguire un determinato compito, come ad esempio una funzione che riceve in input una stringa e la stampa sullo schermo. Proprio per questo motivo alcuni linguaggi di programmazione supportano due tipi di sottoprogrammi; abbiamo visto ad esempio che il **FORTRAN** distingue tra **FUNCTION** e **SUBROUTINE**. La **FUNCTION** si comporta proprio come una funzione matematica ed ha sempre un valore di ritorno; la **SUBROUTINE** invece e' un semplice sottoprogramma che esegue un determinato compito senza produrre nessun valore in output.

La struttura di una procedura Assembly.

Dopo aver illustrato gli aspetti teorici, passiamo agli aspetti pratici e analizziamo il procedimento che bisogna seguire per creare un sottoprogramma in Assembly; nella sua forma piu' semplice un sottoprogramma Assembly e' formato da un nome simbolico, da una direttiva **PROC** (procedure), da un corpo, da un'istruzione **RET** (return from procedure) e nuovamente dal nome simbolico seguito dalla direttiva **ENDP** (end of procedure). Possiamo quindi schematizzare lo scheletro di un sottoprogramma Assembly chiamato **ProcTest** in questo modo:

```
ProcTest PROC

   ret

ProcTest ENDP
```

Come si puo' notare, in questa struttura non c'e' traccia ne della lista dei parametri ne di eventuali valori di ritorno; e' chiaro infatti che in Assembly tutti questi aspetti sono a carico del programmatore. Come e' stato spiegato nei precedenti capitoli, l'Assembly non e' un vero linguaggio di programmazione, ma e' un insieme di strumenti attraverso i quali e' possibile fare un po di tutto; i linguaggi di alto livello utilizzano questi strumenti dell'Assembly per implementare i meccanismi che consentono ad un sottoprogramma di ricevere argomenti o di restituire i valori di ritorno. Allo stesso modo, un programmatore Assembly e' libero di implementare un proprio sistema per inviare argomenti ad un sottoprogramma e per ricevere valori di ritorno dal sottoprogramma stesso; tutti questi aspetti vengono illustrati piu' avanti in questo stesso capitolo.
Ci si puo' chiedere quale possa essere l'area di un programma Assembly piu' adatta a contenere le varie procedure; sicuramente la scelta migliore e' quella illustrata nello scheletro di un programma Assembly presentato nel Capitolo 12 (**Struttura di un programma Assembly**). Naturalmente siamo liberi di scegliere anche altre soluzioni; nel caso ad esempio dei programmi in formato **EXE** e' anche possibile seguire lo stile **Pascal**, inserendo le procedure prima dell'entry point del programma. L'aspetto fondamentale da tener presente e' dato dal fatto che una procedura deve essere opportunamente separata dal programma principale; inoltre e' necessario prendere tutte le precauzioni per evitare che nel corso dell'esecuzione di un programma ci si possa ritrovare inavvertitamente all'interno di una procedura. Proprio per questo motivo e' importante sistemare le procedure prima dell'entry point o dopo l'exit point; al momento opportuno vedremo come affrontare questi problemi e anche come creare apposite librerie esterne di procedure.

Procedure e trasferimento del controllo.

In relazione alle procedure, l'argomento piu' importante che dobbiamo affrontare riguarda sicuramente il meccanismo attraverso il quale avviene la chiamata delle procedure stesse; tutti questi aspetti sono stati illustrati in modo dettagliato nel Capitolo 19 (**Istruzioni per il trasferimento del controllo**). Ricapitoliamo i concetti piu' importanti.

Come accade per le variabili, anche le procedure hanno un indirizzo di memoria; l'indirizzo di una procedura rappresenta l'indirizzo iniziale del blocco di memoria che contiene il corpo della procedura stessa. Supponiamo ad esempio di definire una procedura chiamata **ProcTest** all'offset **00B2h** del segmento di codice **CODESEGM**; in questo caso, l'indirizzo logico di **ProcTest** e' rappresentato dalla coppia **CODESEGM:00B2h**. Questo e' anche l'indirizzo logico relativo al punto in cui viene definito il nome **ProcTest**; tenendo conto del fatto che il nome di una procedura e' una semplice etichetta che delimita l'inizio della procedura stessa, possiamo anche dire che la coppia **CODESEGM:00B2h** rappresenta l'indirizzo logico della prima istruzione di **ProcTest**. In definitiva, l'istruzione:

```
mov bx, offset ProcTest,
```
carichera' in **BX** il valore **00B2h**; analogamente l'istruzione:
```
mov dx, seg ProcTest,
```
carichera' in **DX** il paragrafo che il **SO** assegna a **CODESEGM** al momento di caricare il programma in memoria.

La chiamata di una procedura viene effettuata attraverso l'istruzione **CALL** (call procedure); la parte del programma che chiama una procedura viene definita **caller** (chiamante). La procedura che viene chiamata viene definita **called** (chiamato); il caller puo' essere sia il programma principale che un'altra procedura. In presenza di un'istruzione **CALL** la CPU salva nello stack l'indirizzo di ritorno, carica in **CS:IP** l'indirizzo logico della procedura da chiamare e trasferisce il controllo (salta) all'istruzione associata a **CS:IP** (prima istruzione della procedura); l'indirizzo di ritorno e' quello dell'istruzione immediatamente successiva alla **CALL**.

Al termine della procedura e' necessario restituire il controllo al caller; questo processo viene attivato dall'istruzione **RET** che e' l'ultima istruzione di una procedura. In presenza di un'istruzione **RET** la CPU estrae dallo stack l'indirizzo di ritorno, lo carica in **CS:IP** e trasferisce il controllo (salta) all'istruzione associata a **CS:IP**; in assenza di errori nella gestione dello stack, questa istruzione e' quella immediatamente successiva alla **CALL**.

Una procedura puo' essere chiamata sia in modo diretto, attraverso il proprio nome, sia in modo indiretto, attraverso il proprio indirizzo; nel primo caso si parla di **direct call** (chiamata diretta), mentre nel secondo caso si parla di **indirect call** (chiamata indiretta).
Una procedura da chiamare puo' trovarsi sia nello stesso segmento di programma del caller, sia in un segmento di programma diverso da quello del caller; nel primo caso si parla di **intrasegment call** (chiamata intrasegmento), mentre nel secondo caso si parla di **intersegment call** (chiamata intersegmento).
Complessivamente si possono presentare quindi quattro casi differenti; questi quattro casi vengono illustrati in dettaglio nei paragrafi seguenti.

Chiamata diretta intrasegmento.

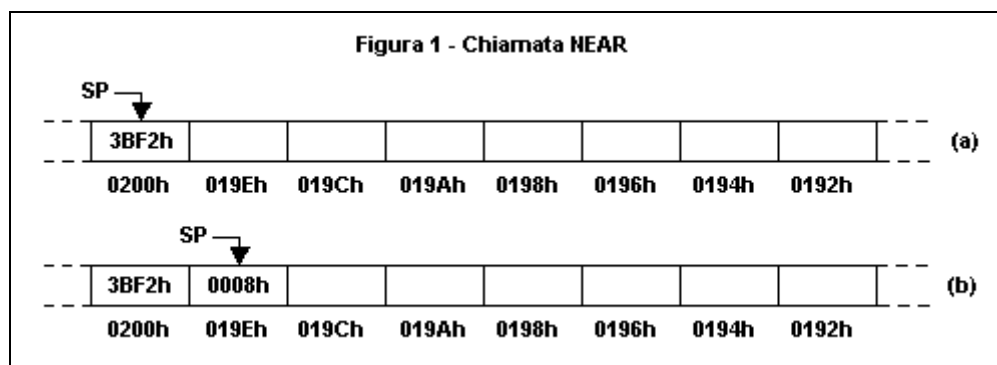
Facciamo riferimento sempre alla nostra procedura **ProcTest** definita all'offset **00B2h** del segmento di codice **CODESEGM**; supponiamo ora che all'interno dello stesso segmento **CODESEGM** siano presenti le seguenti istruzioni:

```
call     near ptr ProcTest    ; 0005h  E8h 00AAh
mov      bx, ax               ; 0008h  8Bh D8h
```

Alla destra di ogni istruzione e' presente un commento contenente l'offset e il codice macchina dell'istruzione stessa; come possiamo notare, all'offset **0005h** del blocco **CODESEGM** e' presente un'istruzione di chiamata diretta intrasegmento alla procedura **ProcTest**. Sia il caller che il called si trovano nello stesso segmento di programma; in un caso del genere la chiamata della procedura viene definita di tipo **NEAR** (near = vicino). Una chiamata **NEAR** consiste quindi in un trasferimento del controllo ad un indirizzo che si trova all'interno dello stesso segmento di programma del caller; la CPU ha bisogno di conoscere solo l'offset dell'indirizzo a cui saltare in quanto il contenuto di **CS** rimane invariato. Per enfatizzare il fatto che si tratta di una chiamata **NEAR**, si puo' utilizzare l'operatore **NEAR PTR** per informare l'assembler che l'indirizzo che segue e' formato da un solo offset; questo operatore diventa necessario quando la chiamata di una procedura precede la definizione della procedura stessa.

Il codice macchina della chiamata diretta **NEAR** e' **E8h**; questo opcode e' seguito da un valore a **16** bit (**00AAh**). Come gia' sappiamo, la CPU somma questo valore all'offset dell'istruzione successiva alla **CALL** per ottenere l'offset a cui saltare; nel nostro caso si ha:

```
00AAh + 0008h = 00B2h,
```

che e' proprio l'offset di **ProcTest**. Prima di tutto la CPU salva nello stack l'offset **0008h** dell'istruzione successiva alla **CALL**; subito dopo la CPU carica l'offset **00B2h** in **IP** e salta a **CS:IP**.

La Figura 1 ci permette di vedere esattamente quello che succede nello stack; la Figura 1a illustra lo stato del segmento di stack prima della chiamata di **ProcTest**, mentre la Figura 1b illustra lo stato del segmento di stack dopo l'esecuzione dell'istruzione **CALL**. Come al solito la memoria viene rappresentata con gli indirizzi crescenti da destra verso sinistra.
Nella Figura 1a vediamo che prima dell'esecuzione dell'istruzione **CALL**, lo stack pointer **SP** contiene l'offset **0200h** relativo al segmento di stack del nostro programma; quest'offset rappresenta l'attuale cima dello stack (**TOS**), e punta ad una word **3BF2h** memorizzata in precedenza dal nostro programma. Al momento di eseguire l'istruzione **CALL**, la CPU sottrae **2** byte ad **SP** per far posto ad una nuova word da memorizzare, e ottiene **SP=019Eh** (Figura 1b); all'indirizzo **SS:SP** (cioe' in **SS:019Eh**) viene salvato l'offset **0008h** dell'indirizzo di ritorno. Subito dopo la CPU carica in **IP** l'offset **00B2h** di **ProcTest** e salta al nuovo **CS:IP**; appena entriamo in **ProcTest**, la coppia **SS:SP** vale quindi **SS:019Eh** e punta alla word **0008h**. Al termine di **ProcTest** e' presente l'istruzione **RET** che attiva il procedimento per la restituzione del controllo al caller; in conseguenza del fatto che la procedura **ProcTest** e' di tipo **NEAR**, l'assembler assegna a **RET** il codice macchina **C3h** del ritorno di tipo **NEAR**. Il ritorno **NEAR** consiste nell'estrazione dallo stack di un valore a **16** bit da caricare in **IP**, mentre **CS** rimane inalterato. Nel nostro caso, se non ci sono stati errori nella gestione dello stack, la CPU estrae da **SS:019Eh** l'offset **0008h**, e ripristina lo stack pointer sommando **2** byte a **SP** (ottenendo **SP=0200h**); l'offset **0008h** viene caricato in **IP** e l'esecuzione del programma salta a **CS:IP** (**CS:0008h**). Come si puo' notare, quando il controllo torna al caller, la coppia **SS:SP** vale **SS:0200h** e punta alla word **3BF2h**; questa e' la stessa identica situazione che caratterizzava il segmento di stack prima dell'esecuzione dell'istruzione **CALL**. Naturalmente tutto cio' si verifica solo se la procedura **ProcTest** non ha commesso errori nella gestione dello stack; in sostanza questo significa che all'interno di **ProcTest** tutti i byte eventualmente inseriti nello stack devono essere perfettamente bilanciati da altrettanti byte estratti dallo stack.

Vediamo un esempio pratico che attraverso la procedura **writeHex16** della libreria **EXELIB** ci permette di visualizzare sullo schermo l'indirizzo di ritorno della procedura **ProcTest**; naturalmente questo indirizzo puo' variare da computer a computer. Il codice di **ProcTest** e' il seguente:

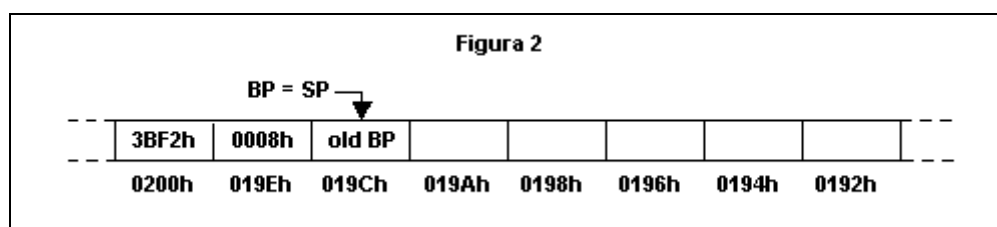
```
ProcTest proc near

   push     bp              ; salva bp
   mov      bp, sp          ; ss:bp = ss:sp

   mov      ax, [bp+2]      ; ax = ss:[bp+2]
   mov      dx, 1000h       ; riga 10h, colonna 00h
   call     writeHex16      ; stampa ax in hex.

   pop      bp              ; ripristina bp
   ret                      ; near return

ProcTest endp
```

Il codice appena scritto merita un'analisi dettagliata; prima di tutto notiamo che e' possibile specificare la modalita' di indirizzamento di un nome simbolico, attraverso operatori come **NEAR** e **FAR**. Nel nostro caso abbiamo scritto:

`ProcTest proc near.`

In questo caso stiamo dicendo all'assembler che **ProcTest** e' una procedura di tipo **NEAR** che verra' quindi chiamata da una **NEAR** call; di conseguenza l'assembler sa che anche l'istruzione **RET** dovra' essere codificata come un **NEAR** return.
Appena entrati in **ProcTest** sappiamo che **SS:SP** punta all'indirizzo di ritorno; se **SS:SP** punta all'indirizzo di ritorno, allora **SS:[SP]** e' l'indirizzo di ritorno . E' proibito pero' dereferenziare **SP** con istruzioni del tipo:

`mov ax, [sp],`

in quanto questa e' una prerogativa riservata solo alla CPU. Al posto di **SP** possiamo usare allora il base pointer **BP**; questo registro (come succede per **SP**) non ha bisogno del segment override in quanto la CPU lo associa automaticamente a **SS**. Per i motivi che vedremo piu' avanti, i compilatori **C** e **Pascal** esigono che le procedure chiamate preservino il contenuto originario di **BP**; seguendo queste convenzioni (anche se in Assembly non siamo obbligati a farlo), prima di usare **BP** salviamo nello stack il suo vecchio contenuto. Successivamante il contenuto di **SP** viene copiato in **BP** in modo che l'indirizzo logico **SS:SP** coincida con l'indirizzo logico **SS:BP**; come si vede in Figura 2, a causa del salvataggio nello stack del vecchio contenuto di **BP**, l'indirizzo di ritorno non si trova piu' a **SS:BP**, ma a **SS:BP+2** (il valore **old BP** indica appunto il vecchio contenuto di **BP** appena salvato nello stack).

Il valore a **16** bit (**0008h**) puntato da **SS:BP+2** viene caricato in **AX** e visualizzato sullo schermo da **writeHex16**; per verificare l'esattezza del valore stampato sullo schermo possiamo consultare il listing file del nostro programma. Prima dell'istruzione **RET**, dobbiamo ripristinare lo stack (e il vecchio contenuto di **BP**) scrivendo:

`pop bp.`

Se dimentichiamo questa istruzione, la CPU incontrando **RET** estrae dallo stack il vecchio

contenuto di **BP** (**old BP**), lo carica in **IP** e salta a **CS:IP**; naturalmente questo non e' l'indirizzo di ritorno corretto, e il nostro programma (generalmente) va in crash.

Chiamata indiretta intrasegmento.

Per la chiamata indiretta intrasegmento si possono fare considerazioni assolutamente analoghe al caso della chiamata diretta intrasegmento; l'unica differenza e' data dal fatto che l'operando di **CALL** questa volta non e' il nome di una procedura **NEAR**, ma e' un valore a **16** bit che rappresenta l'offset della procedura **NEAR** da chiamare. Utilizzando sempre l'esempio della procedura **ProcTest** definita all'offset **00B2h** del blocco **CODESEGM**, possiamo scrivere:

```
mov      dx, offset ProcTest     ; 0005h  BAh 00B2h
call     word ptr dx             ; 0008h  FFh D2h
mov      bx, ax                  ; 000Ah  8Bh D8h
```

Come si puo' notare, in questo caso e' stato utilizzato il regitro **DX** per contenere l'offset di **ProcTest**; si puo' anche utilizzare qualsiasi altro registro generale a **16** bit. L'istruzione **CALL** questa volta ha il codice macchina **FFh** che indica una chiamata indiretta (la chiamata intrasegmento e' codificata nel secondo opcode); per enfatizzare il fatto che l'operando di **CALL** e' un valore a **16** bit, si puo' utilizzare l'operatore **WORD PTR**. In sostanza, anche in questo caso la chiamata e' di tipo **NEAR**; al termine di **ProcTest**, l'istruzione **RET** attivera' di conseguenza un ritorno di tipo **NEAR**. In base a tutte queste considerazioni, possiamo dire che in relazione alla chiamata di **ProcTest**, la gestione dello stack da parte della CPU e' assolutamente identica al caso della chiamata diretta intrasegmento.
In alternativa ai registri generali, per contenere l'offset di **ProcTest** si puo' anche utilizzare una qualsiasi variabile a **16** bit; questo caso e' stato illustrato in dettaglio nel Capitolo 19. A titolo di curiosita' vediamo come ci si deve comportare quando si vuole utilizzare una variabile definita in un segmento dati diverso da quello referenziato da **DS**; supponiamo che all'offset **0000h** di un blocco dati chiamato **DATASEGM2** sia presente la seguente definizione:

```
procAddr dw 0.
```

Se vogliamo chiamare indirettamente **ProcTest** attraverso **procAddr** possiamo scrivere ad esempio:

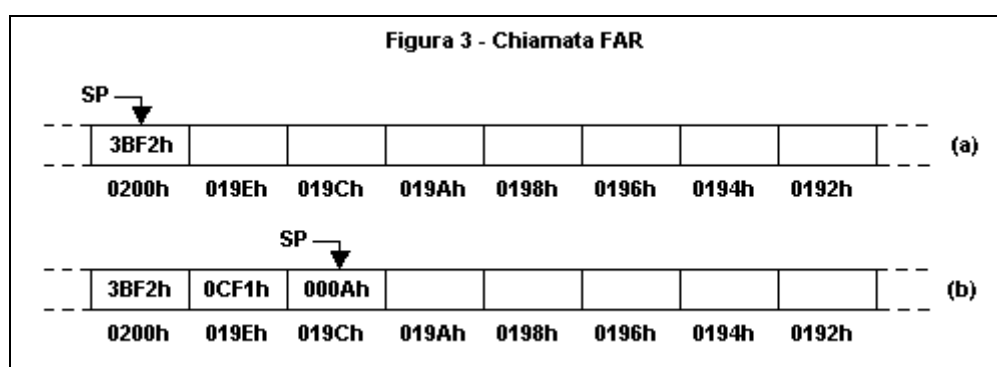
```
mov      ax, DATASEGM2                   ; 0005h  B8h 0000h
mov      es, ax                          ; 0008h  8Eh C0h
mov      es:procAddr, offset ProcTest    ; 000Ah  26h C6h 07h 0000h 00B2h
call     word ptr es:procAddr            ; 0011h  26h FFh 16h 0000h
mov      bx, ax                          ; 0016h  8Bh D8h
```

Come gia' sappiamo, rinunciando ad usare la direttiva **ASSUME** per associare **DATASEGM2** a **ES**, dobbiamo provvedere noi stessi ad inserire i necessari segment override; nel codice macchina notiamo appunto la presenza del valore **26h** che e' proprio il segment override per il registro **ES**. Se avessimo utilizzato la direttiva **ASSUME**, questi segment override sarebbero stati inseriti dall'assembler; quando la CPU incontra il codice macchina **26h**, sa che deve calcolare gli offset dei dati rispetto a **ES** e non rispetto a **DS**.
Riassumendo, nella chiamata diretta intrasegmento si consiglia di far precedere l'operando di **CALL** dall'operatore **NEAR PTR**; nella chiamata indiretta intrasegmento si consiglia di far precedere l'operando di **CALL** dall'operatore **WORD PTR**. Queste precauzioni, anche se in molti casi non sono necessarie, consentono di rendere il codice piu' chiaro e possono anche aiutare l'assembler a generare il codice macchina piu' opportuno.

Chiamata diretta intersegmento.

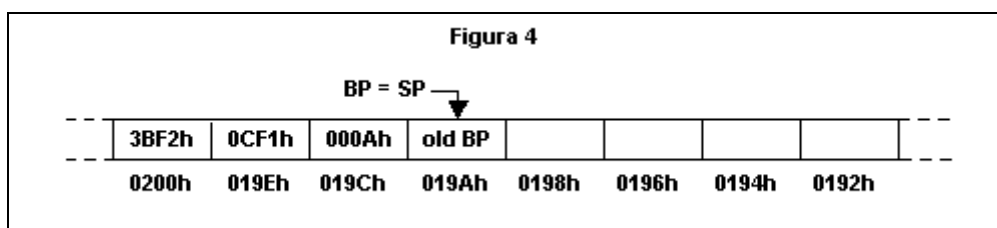
La chiamata diretta intersegmento si verifica quando chiamiamo direttamente (per nome) una procedura definita in un segmento di programma differente da quello in cui si trova il caller; supponiamo quindi che il caller si trovi nel blocco **CODESEGM**, mentre la solita procedura **ProcTest** (called) sia definita all'offset **00B2h** del blocco **CODESEGM2**. Consideriamo allora le seguenti istruzioni presenti nel blocco **CODESEGM**:

```
call      far ptr ProcTest       ; 0005h  9Ah 000000B2h
mov       bx, ax                 ; 000Ah  8Bh D8h
```

All'offset **0005h** del blocco **CODESEGM** e' presente un'istruzione di chiamata diretta intersegmento alla procedura **ProcTest**; il caller si trova in **CODESEGM** mentre il called si trova in **CODESEGM2**. In un caso del genere la chiamata della procedura viene definita di tipo **FAR** (far = lontano); in generale una chiamata **FAR** consiste in un trasferimento del controllo ad un indirizzo logico formato da una coppia **seg:offset**. In sostanza, la CPU ha bisogno di conoscere non solo l'offset, ma anche il segmento dell'indirizzo logico a cui saltare; nel caso del nostro esempio il controllo passa da **CODESEGM:0005h** a **CODESEGM2:00B2h**. Per enfatizzare il fatto che si tratta di una chiamata **FAR**, si puo' utilizzare l'operatore **FAR PTR** per informare l'assembler che l'indirizzo che segue e' formato da una coppia **seg:offset**; questo operatore diventa necessario quando la chiamata di una procedura precede la definizione della procedura stessa.
Il codice macchina della chiamata diretta **FAR** e' **9Ah**; questo opcode e' seguito da un valore a **32** bit (**000000B2h**) che deve essere interpretato come una coppia **seg:offset**. La componente **offset** occupa i **16** bit meno significativi, mentre la componente **seg** occupa i **16** bit piu' significativi; naturalmente, per la componente **seg** l'assembler usa un valore simbolico **0000h**, in quanto il vero valore sara' noto solo al momento di caricare il programma in memoria.
In virtu' del fatto che la **FAR** call modifica anche **CS**, prima di effettuare la chiamata la CPU salva nello stack la coppia **CODESEGM:000Ah** relativa all'istruzione successiva alla **CALL** (indirizzo di ritorno); subito dopo la CPU carica la coppia **CODESEGM2:00B2h** in **CS:IP** e salta a **CS:IP**.

La Figura 3 ci permette di vedere esattamente quello che succede nello stack; la Figura 3a illustra lo stato del segmento di stack prima della chiamata di **ProcTest**, mentre la Figura 3b illustra lo stato del segmento di stack dopo l'esecuzione dell'istruzione **CALL**.
Nella Figura 3a vediamo che prima dell'esecuzione dell'istruzione **CALL**, lo stack pointer **SP** contiene l'offset **0200h** relativo al segmento di stack del nostro programma; quest'offset rappresenta l'attuale cima dello stack (**TOS**), e punta ad una word (**3BF2h**) memorizzata in precedenza dal nostro programma. Al momento di eseguire l'istruzione **CALL**, la CPU sottrae **2** byte ad **SP** per far posto ad una nuova word da memorizzare, e ottiene **SP=019Eh** (Figura 3b); all'indirizzo **SS:SP** (cioe' in **SS:019Eh**) viene salvato il paragrafo assegnato a **CODESEGM** dal **SO** (nel nostro esempio supponiamo di avere **CODESEGM = 0CF1h**). In seguito la CPU sottrae altri **2** byte ad **SP**

per far posto ad una nuova word da memorizzare, e ottiene **SP=019Ch** (Figura 3b); all'indirizzo **SS:SP** (cioe' in **SS:019Ch**) viene salvato l'offset **000Ah** dell'indirizzo di ritorno. Come si puo' notare, la CPU dispone sempre le coppie **seg:offset** in modo che la componente **offset** preceda la componente **seg**; come gia' sappiamo, questa convenzione e' estremamente importante e deve essere rigorosamente seguita anche dal programmatore.
Subito dopo aver salvato nello stack l'indirizzo di ritorno, la CPU carica in **CS:IP** la coppia **CODESEGM2:00B2h** relativa a **ProcTest** e salta al nuovo **CS:IP**; appena entriamo in **ProcTest**, la coppia **SS:SP** vale quindi **SS:019Ch** e punta alla word **000Ah**, mentre la coppia **SS:SP+2** vale **SS:019Eh** e punta alla word **0CF1h**.
Al termine di **ProcTest** e' presente l'istruzione **RET** che attiva il procedimento per la restituzione del controllo al caller; in conseguenza del fatto che la procedura **ProcTest** e' di tipo **FAR**, l'assembler assegna a **RET** il codice macchina **CBh** del ritorno di tipo **FAR**. Ricordiamo che in in questo caso, per rendere esplicito il fatto che la procedura e' dotata di **FAR** return, al posto di **RET** si puo' utilizzare l'istruzione equivalente **RETF**. Il ritorno **FAR** consiste nell'estrazione dallo stack di due valori a **16** bit da caricare in **CS:IP**; la prima word estratta viene caricata in **IP**, mentre la seconda word estratta viene caricata in **CS**. Come al solito, se non ci sono stati errori nella gestione dello stack, la CPU estrae dallo stack la coppia **0CF1h:000Ah**, e ripristina lo stack pointer sommando **4** byte a **SP** (ottenendo **SP=0200h**); la coppia **0CF1h:000Ah** viene caricata in **CS:IP** e l'esecuzione del programma salta a **CS:IP**. Come si puo' notare, quando il controllo torna al caller, la coppia **SS:SP** vale **SS:0200h** e punta alla word **3BF2h**; questa e' la stessa identica situazione che caratterizzava il segmento di stack prima dell'esecuzione dell'istruzione **CALL**. Anche in questo caso quindi, bisogna ribadire il fatto che se la procedura **ProcTest** utilizza lo stack, deve farlo in modo corretto, equilibrando perfettamente gli inserimenti con le estrazioni.
A questo punto sarebbe interessante ripetere l'esempio pratico visto per la chiamata diretta intrasegmento; bisogna tener presente pero' che le definizioni di tutte le procedure della libreria **EXELIB** si trovano nel file **EXELIB.OBJ** all'interno di un segmento di codice chiamato **CODESEGM**. Tutte le procedure inoltre sono state dichiarate di tipo **NEAR**; questo significa che possono essere chiamate solo con **NEAR** calls da un caller che si trova in un'altro segmento **CODESEGM**. In un'altro capitolo vedremo come superare questa limitazione scrivendo apposite librerie di procedure **NEAR** e **FAR**; per il momento possiamo aggirare il problema in modo molto semplice. Chiediamo alla procedura **ProcTest** di restituirci in due appositi registri le due componenti **seg** e **offset** dell'indirizzo di ritorno; questi due valori vengono ricevuti dal caller che trovandosi in **CODESEGM** puo' passarli alle procedure di **EXELIB**. Per i valori di ritorno utilizziamo i due registri **SI** (seg) e **DI** (offset); il blocco **CODESEGM2** assume quindi il seguente aspetto:

```
CODESEGM2   SEGMENT  PARA PUBLIC USE16 'CODE'

   ASSUME   cs: CODESEGM2  ; assegna CODESEGM2 a CS

ProcTest proc far

   push     bp             ; salva bp
   mov      bp, sp         ; ss:bp = ss:sp

   mov      si, [bp+4]     ; si = ss:[bp+4] (seg)
   mov      di, [bp+2]     ; di = ss:[bp+2] (offset)

   pop      bp             ; ripristina bp
   ret                     ; far return

ProcTest endp

CODESEGM2   ENDS
```

Prima di tutto notiamo che la procedura **ProcTest** questa volta viene dichiarata di tipo **FAR**; questo significa che il nome **ProcTest** puo' essere chiamato solo con una **FAR** call.
Come e' stato detto in precedenza, una **FAR** call consiste in un salto ad un indirizzo che deve comprendere sia la componente **offset** che la componente **seg**; di conseguenza, per chiamare una procedura **FAR** dobbiamo usare sempre una **FAR** call anche se il caller si trova all'interno dello stesso segmento della procedura. Osserviamo infatti che una procedura **FAR** e' dotata di **FAR** return; se proviamo a chiamare questa procedura con una **NEAR** call, il programma va in crash a causa dell'incompatibilita' tra **NEAR** call e **FAR** return.
Un'altro aspetto importante da osservare riguarda il fatto che all'inizio di ogni segmento in cui sono presenti istruzioni, dobbiamo ricordarci di inserire l'apposita direttiva **ASSUME**; in questo modo stiamo indicando all'assembler quale sara' il paragrafo da caricare in **CS** al momento di effettuare un salto **FAR** ad un nuovo segmento. In modalita' reale, un qualunque segmento di programma puo' essere contemporaneamente leggibile, scrivibile ed eseguibile; nessuno allora ci impedisce di inserire le procedure anche nei segmenti di dati. Supponendo ad esempio di voler inserire procedure nel blocco **DATASEGM**, dobbiamo ricordarci di inserire nello stesso blocco anche la direttiva:
`ASSUME cs: DATASEGM`.
Tornando alla nostra procedura **ProcTest**, supponiamo di voler accedere allo stack tramite **BP**; seguendo le convenzioni **C** e **Pascal**, preserviamo il contenuto originario di **BP** salvandolo nello stack. Subito dopo questo nuovo inserimento e dopo aver assegnato a **BP** il contenuto di **SP**, lo stack assume l'aspetto mostrato in Figura 4:

La componente **offset** dell'indirizzo di ritorno si trova quindi a **SS:BP+2**, mentre la componente **seg** si trova a **SS:BP+4**; queste due componenti vengono salvate rispettivamente nei registri **DI** e **SI** per essere inviate al caller. Come al solito, prima dell'istruzione **RET**, dobbiamo ripristinare lo stack (e il vecchio contenuto di **BP**) scrivendo:
`pop bp`.

A questo punto, viene incontrato un **FAR** return che estrae dallo stack la coppia **0CF1h:000Ah** la carica in **CS:IP** e salta a **CS:IP** restituendo in questo modo il controllo al caller; il blocco **CODESEGM** contenente il caller, assume il seguente aspetto:

```
CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   ASSUME   cs: CODESEGM            ; assegna CODESEGM a CS

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   call     far ptr ProcTest        ; 0005h  9Ah 000000B2h
   mov      ax, si                  ; 000Ah  8Bh C6h
   mov      dx, 2000h               ; 000Ch  BAh 20000h
```

```
   call     writeHex16             ; 000Fh  E8h 0000h
   mov      ax, di                 ; 0012h  8Bh C7h
   inc      dh                     ; 0014h  FEh C6h
   call     writeHex16             ; 0016h  E8h 0000h

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                ; codice di uscita
   mov      ah, 4ch                ; proc. return to DOS
   int      21h                    ; chiama i servizi DOS

CODESEGM    ENDS
```

Al termine di **ProcTest** quindi, il caller riceve le componenti **seg** e **offset** dell'indirizzo di ritorno nei due registri **SI** e **DI**; questi due valori vengono poi stampati sullo schermo attraverso **writeHex16**. La chiamata a **writeHex16** e' possibile in quanto sia il caller che la procedura si trovano nello stesso blocco **CODESEGM**; in questo caso particolare non e' necessario l'operatore **NEAR PTR** in quanto l'include file **EXELIB.INC** contiene la dichiarazione:
`EXTRN writeHex16: NEAR`.
Questa dichiarazione inoltre precede la chiamata a **writeHex16** per cui l'assembler sa gia' che la chiamata e' di tipo **NEAR**; tutti questi aspetti verranno illustrati in dettaglio in un apposito capitolo.

Chiamata indiretta intersegmento.

La chiamata indiretta intersegmento e' assolutamente analoga alla chiamata diretta intersegmento; l'unica differenza e' rappresentata dal fatto che l'operando di **CALL** non e' il nome di una procedura **FAR**, ma e' l'indirizzo completo (**seg:offset**) di una procedura **FAR**.
Ripetiamo l'esempio precedente con una chiamata indiretta intersegmento; a tale proposito definiamo nel blocco **DATASEGM** una variabile a **32** bit chiamata **procAddr**. Questa variabile dovra' contenere l'indirizzo completo **seg:offset** di **ProcTest**; nel rispetto della convenzione piu' volte citata in precedenza, la componente **offset** deve occupare i **16** bit meno significativi di **procAddr**, mentre la componente **seg** deve occupare i **16** bit piu' significativi di **procAddr**. La **FAR** call indiretta a **ProcTest** puo' essere allora riscritta in questo modo:

```
mov      word ptr procAddr[0], offset ProcTest
mov      word ptr procAddr[2], seg ProcTest
call     dword ptr procAddr
```

Le componenti **seg** e **offset** di **ProcTest** sono due valori a **16** bit; per copiare questi valori nella variabile **procAddr** a **32** bit, bisogna spezzare **procAddr** in due meta' attraverso l'operatore **WORD PTR**. La scritta:
`word ptr procAddr[0],`
rappresenta il contenuto dei due byte di **procAddr** che si trovano nelle posizioni **0** e **1** (prima word); la scritta:
`word ptr procAddr[2],`
rappresenta il contenuto dei due byte di **procAddr** che si trovano nelle posizioni **2** e **3** (seconda word).
Per enfatizzare il fatto che l'operando di **CALL** e' una variabile a **32** bit, possiamo utilizzare l'operatore **DWORD PTR**; questo operatore e' superfluo se la definizione di **ProcTest** precede la sua chiamata.
Riassumendo, nella chiamata diretta intersegmento si consiglia di far precedere l'operando di **CALL** dall'operatore **FAR PTR**; nella chiamata indiretta intersegmento si consiglia di far precedere l'operando di **CALL** dall'operatore **DWORD PTR**.

E' importante ricordare che in modalita' reale, per la chiamata indiretta intersegmento non e' possibile utilizzare un registro a **32** bit scrivendo ad esempio:

```
mov      ax, seg ProcTest     ; carica seg in AX
shl      eax, 16              ; sposta seg nella MSW di EAX
mov      ax, offset ProcTest  ; carica offset in AX
call     eax                  ; chiamata indiretta inters.
```

I registri a **32** bit possono svolgere il ruolo di operandi di **CALL** solo in modalita' protetta a **32** bit; in questa modalita' operativa delle CPU **80386** e superiori, gli offset sono appunto valori a **32** bit relativi a segmenti di memoria da **4** Gb.

Considerazioni generali sulle procedure.

All'inizio di questo capitolo sono stati illustrati una serie di benefici legati all'utilizzo delle procedure nei programmi; abbiamo visto in sostanza che scomponendo un programma in tanti sottoprogrammi, si perviene ad uno stile di programmazione modulare che presenta diversi risvolti positivi. Analizzando pero' le cose appena dette in relazione ai metodi di chiamata di una procedura, possiamo mettere in evidenza anche un'aspetto negativo; come si puo' facilmente intuire, questo aspetto negativo e' legato al fatto che ricorrere alle procedure significa inserire nei programmi una serie di istruzioni di salto necessarie per trasferire il controllo dal caller al called e viceversa.
Ciascun salto comporta una certa perdita di tempo dovuta alle ragioni esposte in questo capitolo e anche nella nota in fondo al Capitolo 19; se il ricorso alle procedure e' massiccio, tutte queste perdite di tempo si sommano tra loro e possono avere gravi conseguenze sulle prestazioni dei programmi. Proprio per questo motivo e' importante (soprattutto in Assembly) non eccedere con il ricorso alle procedure; un uso ridotto delle procedure comporta una maggiore sequenzialita' delle istruzioni di un programma con conseguente aumento della velocita' di esecuzione.
Un'altro aspetto abbastanza evidente e' legato al fatto che una **FAR** call e' sicuramente piu' lenta di una **NEAR** call; osserviamo infatti che la **FAR** call comporta una maggiore perdita di tempo dovuta al maggior numero di informazioni da gestire nello stack. La **NEAR** call comporta l'inserimento nello stack di un solo offset a **16** bit, e il conseguente **NEAR** return comporta l'estrazione dallo stack dello stesso offset a **16** bit; la **FAR** call invece comporta l'inserimento nello stack di una coppia **seg:offset** a **16+16** bit, e il conseguente **FAR** return comporta l'estrazione dallo stack della stessa coppia **seg:offset** a **16+16** bit.
I puristi dell'Assembly arrivano addirittura ad evitare del tutto le procedure, sostituendole eventualmente con le macro; nel precedente capitolo infatti, abbiamo visto che le macro vengono espanse nel punto esatto nel quale vengono chiamate, garantendo in questo modo un'ottima sequenzialita' delle istruzioni. In pratica, e' come se il codice completo di una procedura venisse copiato nel punto in cui e' presente la chiamata alla procedura stessa; in questo modo si migliorano le prestazioni del programma in quanto si evita il salto dal caller al called e viceversa.
Naturalmente, sappiamo che anche un uso eccessivo delle macro porta a delle conseguenze negative; ogni chiamata di una macro si traduce infatti nell'espansione del corpo della macro stessa con conseguente aumento delle dimensioni del codice.
Da tutte queste considerazioni si deduce che se vogliamo scrivere programmi compatti e veloci, dobbiamo trovare un buon equilibrio tra modularita' e sequenzialita'.

Passaggio di argomenti alle procedure.

Programmare in Assembly significa avere il privilegio di poter accedere in modo diretto a tutto l'hardware installato nel nostro computer; in questo modo e' possibile sfruttare le tecniche di programmazione piu' potenti ed efficienti, a patto naturalmente di avere una profonda conoscenza dell'architettura del computer. L'Assembly in particolare ci permette di accedere direttamente ai

registri della CPU, e questo e' sicuramente un grande vantaggio per il programmatore; come gia' sappiamo infatti, i registri della CPU sono delle vere e proprie mini-memorie **RAM** statiche, che grazie all'utilizzo dei **flip-flop** ad altissime prestazioni, risultano molto piu' veloci dell'ordinaria memoria **RAM** dinamica del computer. Una diretta conseguenza di questo aspetto, e' data dal fatto che, nei limiti del possibile, il programmatore dovrebbe utilizzare sempre operandi di tipo **reg** per le istruzioni della CPU; in questo modo si ottiene un notevole aumento delle prestazioni dei programmi rispetto al caso in cui si utilizzino operandi di tipo **mem** o **imm**.
Si potrebbe pensare allora di utilizzare i registri della CPU per implementare un metodo velocissimo per il passaggio di eventuali parametri alle procedure; questo e' proprio il metodo impiegato da tutte le procedure delle librerie **EXELIB** e **COMLIB**.
Naturalmente e' anche possibile utilizzare i registri della CPU per gli eventuali valori restituiti dalle procedure (valori di ritorno); vediamo un semplice esempio pratico.
Supponiamo di voler scrivere una procedura chiamata **Somma** che riceve in input due word e produce in output la loro somma; utilizziamo ad esempio **CX** e **DX** per contenere i due addendi, e **AX** per contenere la somma. Siccome si tratta solo di un esempio, trascuriamo il controllo degli errori legati ad eventuali riporti o overflow; otteniamo allora:

```
; ax = Somma(cx, dx)

Somma proc

   mov      ax, cx   ; ax = primo addendo
   add      ax, dx   ; ax = somma

   ret               ; return ax

Somma endp
```

Naturalmente questa procedura presuppone che il programmatore sappia che i due addendi da sommare devono essere passati attraverso **CX** e **DX**, e che la somma finale viene restituita in **AX**; come e' stato detto all'inizio del capitolo, queste informazioni rappresentano l'**interfaccia** della procedura **Somma**.
A questo punto, nel blocco codice del nostro programma possiamo scrivere:

```
mov      cx, 1250       ; primo addendo
mov      dx, 3800       ; secondo addendo
call     Somma          ; ax = cx + dx
```

La procedura **Somma** termina restituendo il risultato finale (valore di ritorno) nel registro **AX**.
Questo semplicissimo esempio ci permette di evidenziare alcuni aspetti importanti legati all'utilizzo dei registri per il passaggio degli argomenti alle procedure; la prima cosa da dire e' relativa al fatto che questa tecnica, per poter essere applicata in modo chiaro, richiede una serie di precise regole. In sostanza, il programmatore deve stabilire quali registri debbano essere usati per il passaggio degli argomenti, e quali registri debbano essere usati per i valori di ritorno; lo scopo di queste regole e' quello di definire un'interfaccia coerente per la chiamata delle procedure. Nel caso ad esempio delle procedure definite nella libreria **EXELIB**, abbiamo visto che vengono utilizzati solamente i tre registri **AX/EAX**, **BX** e **DX** che svolgono ruoli ben precisi; in particolare, il registro **BX** viene sempre utilizzato per passare l'offset di una stringa, mentre il registro **DX** passa alla procedura sempre le coordinate (riga, colonna) dello schermo a partire dalle quali viene stampato l'output. Il registro accumulatore viene invece sempre utilizzato per passare numeri interi a **8**, **16** e **32** bit; tutte le procedure che restituiscono valori interi a **8**, **16** e **32** bit utilizzano ugualmente l'accumulatore.
Queste regole aiutano il programmatore ad utilizzare in modo semplice le procedure di una libreria; in assenza di regole coerenti invece, ogni volta che vogliamo chiamare una procedura appartenente

ad una libreria, dobbiamo consultare la relativa documentazione per poter sapere quali registri si devono utilizzare.

Abbiamo visto quindi che l'uso dei registri della CPU per l'interfacciamento con le procedure, garantisce una elevata velocita' di esecuzione; esistono pero' anche delle "controindicazioni" abbastanza importanti. Osserviamo innanzi tutto che i registri della CPU, essendo disponibili in numero molto limitato, devono essere trattati dal programmatore come risorse estremamente preziose; usare i registri per interfacciarsi con le procedure, significa sottrarli alle altre istruzioni che ne hanno bisogno. Molto spesso questo problema puo' arrivare a pregiudicare in modo grave l'efficienza del programma che stiamo scrivendo; in particolare questo caso si verifica quando abbiamo la necessita' di scrivere procedure che richiedono un numero elevato di argomenti, e che quindi possono arrivare ad impegnare quasi tutti i registri della CPU.
Un'altro problema particolarmente grave, e' dato dal fatto che l'uso dei registri per passare gli argomenti alle procedure, rende praticamente impossibile l'implementazione della ricorsione; questo aspetto viene illustrato in dettaglio nel prossimo capitolo.
In definitiva, l'utilizzo dei registri per l'interfacciamento con le procedure, ha il vantaggio di garantire prestazioni molto elevate; questo vantaggio pero' viene annullato dai numerosi svantaggi, alcuni dei quali piuttosto gravi. Proprio per questo motivo, i moderni linguaggi di programmazione passano i parametri alle procedure attraverso un metodo completamente diverso; questo metodo viene illustrato nei paragrafi seguenti. Prima pero' dobbiamo affrontare una questione molto importante, legata alle due diverse modalita' di passaggio degli argomenti ad una procedura.

Passaggio degli argomenti per valore e per indirizzo.

Gli argomenti possono essere passati ad una procedura attraverso due modalita' chiamate: **passaggio per valore** e **passaggio per indirizzo**; analizziamole in dettaglio.
Passare un argomento per valore significa passare alla procedura una copia del valore dell'argomento; in questo modo la procedura puo' manipolare questa copia senza produrre nessuna conseguenza sull'originale. Vediamo un esempio pratico; supponiamo che nel blocco dati del nostro programma sia presente la seguente definizione di una variabile chiamata **Var1**:

```
Var1 dw 1500.
```

Supponiamo ora di avere una procedura chiamata **Raddoppia**, che attraverso **AX** richiede un valore intero a **16** bit, e sempre attraverso **AX**, restituisce questo valore raddoppiato; se vogliamo passare il contenuto di **Var1** a **Raddoppia** possiamo scrivere:

```
mov      ax, Var1     ; ax = 1500
call     Raddoppia    ; chiama Raddoppia(1500)
```

All'interno di **Raddoppia**, il contenuto (**1500**) di **AX** viene raddoppiato con un'istruzione del tipo:

```
add ax, ax,
```

in modo che la procedura termini restituendo **AX = 3000**; questa modifica del contenuto di **AX** non ha nessuna ripercussione sul contenuto di **Var1** che continua a valere **1500**. E' chiaro infatti che il contenuto di **AX** non ha niente a che vedere con il contenuto della locazione di memoria identificata da **Var1**; in sostanza, attraverso **AX** la procedura **Raddoppia** riceve una copia del valore (**1500**) di **Var1**, e si dice quindi che l'argomento **Var1** e' stato passato per **valore** a **Raddoppia**.

Passare un argomento per indirizzo significa passare alla procedura una copia dell'indirizzo dell'argomento; in questo modo la procedura viene a conoscere l'indirizzo di memoria in cui si trova l'argomento, ed ha quindi la possibilita' di accedere a quell'indirizzo modificandone il contenuto originale. Vediamo un esempio pratico che si riferisce sempre alla variabile **Var1**; supponiamo che questa volta la procedura **Raddoppia** attraverso **BX** richieda l'indirizzo di una variabile contenente un valore intero a **16** bit. Questo valore viene raddoppiato e restituito come al solito in **AX**; la chiamata a **Raddoppia** diventa:

```
mov      bx, offset Var1
call     Raddoppia
```

La struttura della procedura raddoppia e' la seguente:

```
Raddoppia proc

   shl      word ptr [bx], 1
   mov      ax, [bx]

   ret

Raddoppia endp
```

L'istruzione:
`shl word ptr [bx], 1,`
fa scorrere di un posto verso sinistra i **16** bit del valore (**1500**) puntato da **BX**; cio' equivale a moltiplicare per due il valore **1500**, trasformandolo in **3000**. Il valore cosi' ottenuto viene copiato in **AX** e la procedura termina con "return value" **AX = 3000**; non ci vuole molto a capire che alla fine si ottiene anche **Val1 = 3000**! Osserviamo infatti che **Raddoppia** conoscendo l'indirizzo di **Var1** puo' accedere a quell'indirizzo modificandone il contenuto; questo accade proprio attraverso l'istruzione **SHL**. Puo' capitare che tutto cio' accada in modo consapevole da parte del programmatore; puo' anche capitare pero' che il corpo della procedura **Raddoppia** sia frutto di una svista da parte del programmatore che magari intendeva scrivere:

```
Raddoppia proc

   mov      ax, [bx]     ; ax = 1500
   shl      ax, 1        ; ax = 3000

   ret

Raddoppia endp
```

Questa seconda versione di **Raddoppia**, copia in **AX** il contenuto (**1500**) di **[BX]**; successivamente viene raddoppiato il contenuto di **AX** senza che questa modifica si ripercuota sul contenuto di **Var1**. Quando **Raddoppia** restituisce il controllo al caller si ottiene: **AX = 3000** e **Var1 = 1500**; in questi due esempi la procedura **Raddoppia** riceve attraverso **BX** una copia dell'indirizzo di **Var1**, e si dice quindi che l'argomento **Var1** e' stato passato per **indirizzo** a **Raddoppia**.

Il meccanismo del passaggio degli argomenti per indirizzo e' molto importante perche' ci permette di affrontare il caso delle procedure che richiedono argomenti di tipo vettore (comprese le stringhe che, come sappiamo, sono vettori di codici ASCII); e' chiaro che il passaggio di un vettore per valore non sembra una cosa molto sensata, soprattutto quando abbiamo a che fare con vettori formati da centinaia di elementi. In un caso del genere, si puo' risolvere la situazione passando appunto il vettore per indirizzo; in sostanza, la procedura riceve l'indirizzo iniziale del vettore, e se necessario anche il numero di elementi del vettore stesso. Vediamo un esempio relativo ad una procedura chiamata **ToUpperCase** che converte una stringa **ASCII** in maiuscolo; prima di tutto definiamo la stringa da convertire:

```
strTest     db    'Stringa Da Convertire'
strLenght   dw    $ - offset strTest
```

Alla variabile **strLenght** viene assegnato un valore che si ottiene sottraendo l'offset di **strLenght** (**$**) dall'offset di **strTest** in modo da ottenere **21** che e' la lunghezza in byte della stringa; a questo punto possiamo scrivere la procedura **ToUpperCase** che riceve in **BX** l'offset di una stringa, e in **CX** la lunghezza in byte della stringa stessa:

```
; ToUpperCase(bx, cx)

ToUpperCase proc

toupper_loop:
   mov      al, [bx]          ; elemento da esaminare
   cmp      al, 'a'           ; confronto con 'a'
   jb       short next_char   ; non e' una minuscola
   cmp      al, 'z'           ; confronto con 'z'
   ja       short next_char   ; non e' una minuscola
   sub      byte ptr [bx], 32 ; converte in maiuscolo
next_char:
   inc      bx                ; prossimo elemento
   loop     toupper_loop      ; controllo loop

   ret

ToUpperCase endp
```

La procedura esamina uno alla volta, tutti gli elementi della stringa; se un elemento vale meno di **'a'** (**97**) o piu' di **'z'** (**122**), allora non rappresenta una lettera minuscola e si passa quindi all'elemento successivo. In caso contrario, si sottrae **32** al valore dell'elemento convertendolo nel codice **ASCII** di una lettera maiuscola; ricordiamo infatti che nel codice **ASCII**, la distanza costante tra una lettera minuscola e il suo equivalente maiuscolo e' pari a **32**. Nel caso ad esempio della lettera **a** risulta:
'a' - 'A' = 32, e quindi: **'A' = 'a' - 32**.
La chiamata alla procedura **ToUpperCase** e' molto semplice:

```
mov      bx, offset strTest
mov      cx, strLenght
call     ToUpperCase
```

Se vogliamo stampare **strTest** con la procedura **writeString**, dobbiamo ricordarci di aggiungere lo zero finale nella definizione della stringa; la procedura **writeString** si aspetta infatti una stringa **C**.

Tornando al caso generale del passaggio degli argomenti per indirizzo, il concetto fondamentale da sottolineare e' che una procedura che conosce l'indirizzo in memoria di un'argomento, puo' anche modificare involontariamente il contenuto di quell'indirizzo; il passaggio degli argomenti per indirizzo rappresenta quindi una situazione molto delicata che richiede una certa attenzione da parte del programmatore.

Convenzione C per il passaggio di argomenti ai sottoprogrammi.

Molti linguaggi di programmazione di alto livello, passano gli argomenti ai sottoprogrammi attraverso un metodo che si basa sull'uso dello stack; in pratica, gli argomenti da passare al sottoprogramma vengono inseriti uno dopo l'altro nello stack, e successivamente avviene il trasferimento del controllo dal caller al called. Il sottoprogramma chiamato puo' leggere i vari argomenti dallo stack secondo il meccanismo gia' illustrato nelle Figure 1, 2, 3 e 4; e' chiaro che per poter applicare questo meccanismo, e' necessario definire apposite convenzioni che stabiliscano l'ordine in base al quale i vari argomenti vengono inseriti nello stack. Le due convenzioni piu'

importanti sono quelle relative al linguaggio **C** e al linguaggio **Pascal**; queste due convenzioni vengono prese come riferimento anche da altri linguaggi di programmazione.

Cominciamo allora con la convenzione del linguaggio **C** ricordando che in questo caso i sottoprogrammi vengono chiamati **funzioni**; la convenzione **C** stabilisce che gli argomenti da passare ad una funzione, debbano essere inseriti nello stack a partire da quello piu' a destra (ultimo argomento). Per illustrare questa convenzione, facciamo riferimento al seguente programma **C**:

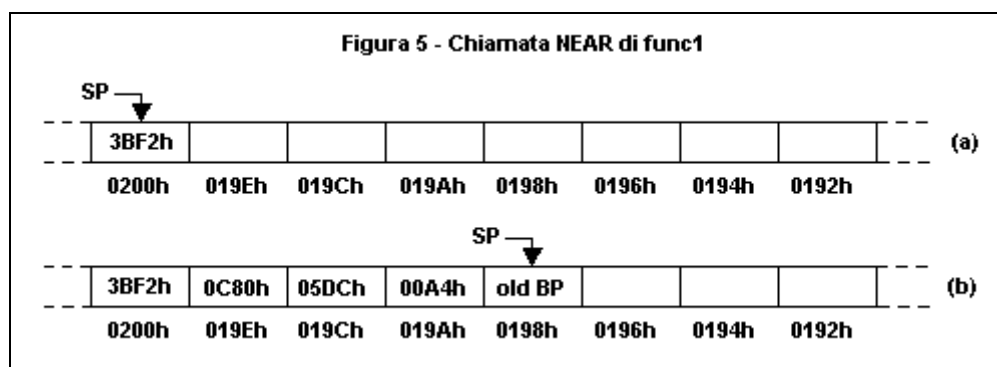
```
/* variabili globali */

int   var1 = 1500;
int   var2 = 3200;

/* prototipi di funzione */

void func1(int, int);

/* entry point */

int main(void)
{
   func1(var1, var2);

   return 0;
}

/* definizione delle funzioni */
.....................
```

Il programma inizia con la definizione di due variabili di tipo **int**, chiamate **var1** e **var2**; nei compilatori **C** a **16** bit per la modalita' reale del **DOS**, il tipo **int** rappresenta un intero con segno a **16** bit che puo' assumere quindi tutti i valori compresi tra **-32768** e **+32767**. Tutte le variabili definite al di fuori di qualsiasi funzione, vengono inserite dal compilatore **C** nel segmento dati del programma; queste variabili quindi sono visibili e accessibili da qualunque funzione del programma.
Successivamente incontriamo il prototipo di una funzione **C** chiamata **func1**; come gia' sappiamo, il prototipo descrive le caratteristiche esterne di una funzione, cioe' l'interfaccia attraverso la quale e' possibile chiamare la funzione. Nel caso del nostro esempio, il prototipo di **func1** afferma che questa funzione richiede due argomenti di tipo **int** e non restituisce nessun valore di ritorno; in **C** il tipo **void** indica appunto il nulla.
A questo punto arriviamo alla funzione **main** che rappresenta l'entry point di tutti i programmi scritti in **C** standard (**ANSI C**); all'interno del **main**, e' presente la chiamata della funzione **func1**, e l'istruzione **return** che termina il programma con exit code **0** (in ambiente **UNIX** un exit code **0** indica la corretta terminazione di un programma).
Subito dopo il **main** iniziano le definizioni delle varie funzioni del programma; ai fini del nostro esempio, ci interessa sapere non com'e' definita **func1**, ma come vengono passati gli argomenti **var1** e **var2** a questa funzione. Cominciamo dal caso in cui la funzione **func1** sia di tipo **NEAR**, e la sua definizione si trovi nello stesso segmento di codice del **main** (caller); in questo caso il compilatore incontrando la chiamata di **func1**, produce il codice macchina corrispondente alle seguenti istruzioni Assembly:

```
push     var2
push     var1
call     near ptr func1
add      sp, 4
```

Come si puo' notare, prima di tutto vengono inseriti nello stack i due argomenti **var1** e **var2** a partire dall'ultimo; successivamente avviene la chiamata di **func1**. Trattandosi di una chiamata di tipo **NEAR**, la CPU inserisce nello stack il solo offset dell'indirizzo di ritorno, carica in **IP** l'offset di **func1** e salta a **CS:IP**; nel nostro esempio, supponiamo che l'indirizzo di ritorno sia **00A4h**. Quest'offset e' relativo chiaramente all'istruzione:

```
add sp, 4;
```

il significato di questa istruzione viene chiarito in seguito. La Figura 5 ci permette di analizzare gli effetti prodotti sullo stack dalla chiamata di **func1**.

Come al solito, supponiamo che inizialmente si abbia **SP=0200h**, con la coppia **SS:SP** (cioe' **SS:0200h**) che punta ad una word (**3BF2h**) inserita in precedenza dal nostro programma (Figura 1a); la fase di chiamata di **func1** determina l'inserimento nello stack di una serie di word. Prima di tutto viene inserito il contenuto (valore) di **var2**; osserviamo che si tratta di un passaggio per valore. Infatti, quando la CPU incontra l'istruzione:

```
push var2,
```

esegue le seguenti operazioni (si suppone che **var1** e **var2** siano definite agli offset **000Ah** e **000Ch** del blocco dati):

1) Sottrae 2 byte a **SP** per fare posto nello stack ad una nuova word, e ottiene **SP=019Eh**.
2) Legge il valore **3200d=0C80h** contenuto nell'indirizzo di memoria **DS:000Ch** di **var2**.
3) Salva questo valore all'indirizzo **SS:SP**, cioe' **SS:019Eh**.

Il valore **0C80h** appena inserito nel segmento di stack, non ha niente a che vedere con il valore **0C80h** di **var2** presente nel segmento di dati; la funzione **func1** ha quindi a disposizione una copia del valore di **var2** e non l'originale.
Il passo successivo consiste nell'inserimento nello stack del valore (**1500**) di **var1**; la CPU esegue i seguenti passi:

1) Sottrae 2 byte a **SP** per fare posto nello stack ad una nuova word, e ottiene **SP=019Ch**.
2) Legge il valore **1500d=05DCh** contenuto nell'indirizzo di memoria **DS:000Ah** di **var1**.
3) Salva questo valore all'indirizzo **SS:SP**, cioe' **SS:019Ch**.

A questo punto incontriamo la chiamata di **func1** che essendo di tipo **NEAR**, determina l'inserimento nello stack dell'offset **00A4h** dell'indirizzo di ritorno; la CPU esegue i seguenti passi:

1) Sottrae 2 byte a **SP** per fare posto nello stack ad una nuova word, e ottiene **SP=019Ah**.
2) Legge l'offset **00A4h** dell'indirizzo di ritorno.
3) Salva quest'offset all'indirizzo **SS:SP**, cioe' **SS:019Ah**.
4) Carica in **IP** l'offset di **func1**.
5) Trasferisce il controllo (salta) all'indirizzo **CS:IP**.

Terminate queste fasi, ci ritroviamo all'interno di **func1**; osservando la Figura 5b possiamo gia' intuire il modo con il quale avviene l'accesso agli argomenti passati alla funzione. Per accedere a questi argomenti viene utilizzato come al solito il Base Pointer **BP**; le prime due istruzioni che il compilatore **C** inserisce in **func1** sono le seguenti:

```
push     bp       ; preserva BP
mov      bp, sp   ; ss:bp = ss:sp
```

Come gia' sappiamo, la prima istruzione salva nello stack il vecchio contenuto di **BP** (che indichiamo con **old BP**); la seconda istruzione copia il contenuto di **SP** in **BP**. Per il salvataggio nello stack del contenuto originale di **BP**, la CPU esegue i seguenti passi:

1) Sottrae 2 byte a **SP** per fare posto nello stack ad una nuova word, e ottiene **SP=0198h**.
2) Legge il contenuto (**old BP**) di **BP**.
3) Salva questo valore all'indirizzo **SS:SP**, cioe' **SS:0198h**.

In base a questa situazione, l'istruzione:
`mov bp, sp,`
copia in **BP** il valore **0198h** contenuto in **SP**; a questo punto, analizzando la Figura 5b possiamo dire che:

1) L'argomento **05DCh** (copia del valore di **var1**) si trova nello stack all'indirizzo **SS:(BP+4)**; infatti:
BP+4 = 019Ch, e quindi **[BP+4] = 05DCh**.
2) L'argomento **0C80h** (copia del valore di **var2**) si trova nello stack all'indirizzo **SS:(BP+6)**; infatti:
BP+6 = 019Eh, e quindi **[BP+6] = 0C80h**.

Questa tecnica quindi consente alle funzioni del linguaggio **C** di accedere ai vari argomenti; al termine di **func1**, il compilatore **C** inserisce le seguenti due istruzioni:

```
pop      bp    ; ripristina BP
ret            ; near return
```

La prima istruzione ripristina il contenuto originale di **BP**; la CPU esegue i seguenti passi:

1) Legge il valore **old BP** dall'indirizzo **SS:SP**, cioe' **SS:0198h**.
2) Somma 2 byte a **SP** per recuperare lo spazio appena liberatosi nello stack, e ottiene **SP=019Ah**.
3) Salva in **BP** il valore **old BP**.

La seconda istruzione (**RET**) rappresenta un **NEAR** return e determina l'estrazione dallo stack dell'offset dell'indirizzo di ritorno; la CPU esegue i seguenti passi:

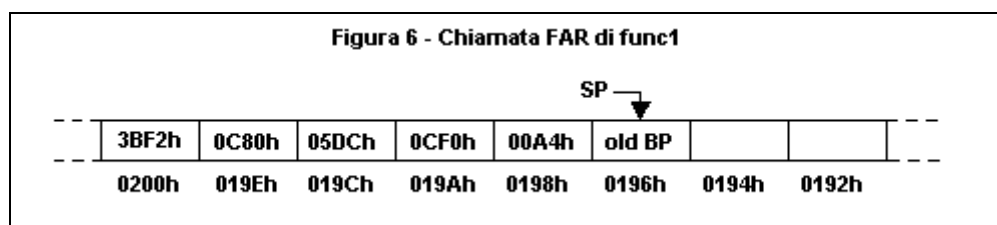
1) Legge il valore **00A4h** dall'indirizzo **SS:SP**, cioe' **SS:019Ah**.
2) Somma 2 byte a **SP** per recuperare lo spazio appena liberatosi nello stack, e ottiene **SP=019Ch**.
3) Salva in **IP** il valore **00A4h**.
4) Trasferisce il controllo (salta) a **CS:IP**.

Attraverso questo salto, il controllo viene restituito al caller; all'indirizzo **CS:IP** (**CS:00A4h**) viene incontrata l'istruzione:
`add sp, 4.`

Come si puo' facilmente intuire, questa istruzione ha il compito di ripulire lo stack dagli argomenti che avevamo passato a **func1**; siccome avevamo inserito due word, pari a **4** byte, sommando **4** a **SP=019Ch** si ottiene **SP=0200h**. A questo punto, la coppia **SS:SP** vale **SS:0200h** e punta alla word **3BF2h**; in questo modo abbiamo perfettamente ripristinato la situazione mostrata in Figura 5a, relativa allo stato dello stack precedente alla chiamata di **func1**.
In base alla convenzione del linguaggio **C**, il compito di ripulire lo stack dagli argomenti ricade sul caller; questa operazione di pulizia puo' essere svolta anche con l'istruzione appena illustrata. Osserviamo che questo e' uno dei rarissimi casi in cui il programmatore ha il compito di modificare il contenuto di **SP**.

Una volta capito il meccanismo della **NEAR** call a **func1**, possiamo affrontare facilmente il caso della **FAR** call; come gia' sappiamo, questo caso si presenta quando la definizione di **func1** si trova in un segmento di programma diverso da quello del caller, oppure, piu' in generale, quando **func1** e' stata dichiarata di tipo **FAR**. Nel caso di **FAR** call, il compilatore **C** genera il codice macchina corrispondente alle seguenti istruzioni Assembly:

```
push     var2
push     var1
call     far ptr func1
add      sp, 4
```

Rispetto alla **NEAR** call, l'unica differenza sta' nel fatto che prima di eseguire l'istruzione **CALL** la CPU salva nello stack, non solo l'offset dell'indirizzo di ritorno, ma anche il contenuto corrente di **CS**; abbiamo visto infatti che per eseguire un salto **FAR** la CPU modifica non solo **IP** ma anche **CS**. In sostanza, per illustrare la situazione che si viene a creare dopo l'esecuzione della **CALL**, ci basta sostituire la precedente Figura 5b con la seguente Figura 6:

In questo esempio supponiamo che il segmento di programma in cui si trova il caller sia **CODESEGM = 0CF0h**; prima di chiamare **func1**, la CPU salva quindi nello stack, prima il contenuto corrente (**0CF0h**) di **CS**, e poi l'offset **00A4h** dell'indirizzo di ritorno. Il valore **0CF0h** viene salvato all'offset **SP=019Ah**, mentre il valore **00A4h** viene salvato all'offset **SP=0198h**. In base a questa situazione, all'interno di **func1** l'istruzione:

`push bp,`

salva il valore corrente (**old BP**) di **BP** all'offset **SP=0196h**; di conseguenza, la successiva istruzione:

`mov bp, sp,`

copia in **BP** il valore **0196h** contenuto in quel momento in **SP**.
A questo punto, osservando la Figura 6 possiamo affermare che:

1) L'argomento **05DCh** (copia del valore di **var1**) si trova nello stack all'indirizzo **SS:(BP+6)**; infatti:
BP+6 = 019Ch, e quindi **[BP+6] = 05DCh**.
2) L'argomento **0C80h** (copia del valore di **var2**) si trova nello stack all'indirizzo **SS:(BP+8)**; infatti:
BP+8 = 019Eh, e quindi **[BP+8] = 0C80h**.

Riassumendo, nel caso di **NEAR** call, la lista degli argomenti inizia da **BP+4**; nel caso invece di **FAR** call, la lista degli argomenti inizia da **BP+6**.

In tutti gli esempi appena illustrati, i vari argomenti sono stati passati per valore a **func1**; il caso relativo ad eventuali argomenti da passare per indirizzo e' altrettanto semplice. Per illustrare questo caso, supponiamo che **func1** richieda il primo argomento per valore e il secondo per indirizzo; il prototipo **C** di **func1** diventa allora:
`void func1(int, *int);`.
Questo prototipo afferma che **func1** richiede due argomenti; il primo e' di tipo **int**, mentre il secondo e' di tipo **indirizzo di un int**. Se vogliamo chiamare **func1** passandole come argomenti il valore di **var1** e l'indirizzo di **var2**, possiamo scrivere:
`func1(var1, &var2);`.
Nel caso ad esempio di **NEAR** call, il compilatore **C** incontrando questa chiamata, genera il codice macchina corrispondente alle seguenti istruzioni Assembly:

```
push     offset var2
push     var1
call     near ptr func1
add      sp, 4
```

L'istruzione:
`push offset var2`,
inserisce nello stack una copia dell'indirizzo (**000Ch**) di **var2**; la funzione **func1** conoscendo questo indirizzo e' in grado di modificare direttamente il contenuto originale di **var2**. Ricordiamo che l'istruzione **PUSH** puo' utilizzare un operando di tipo **imm** solo con le CPU **80286** o superiori; nel caso di CPU **8086** bisogna prima copiare l'operando **imm** in un registro, e poi usare lo stesso registro come operando di **PUSH** (al posto del registro si puo' anche usare un operando di tipo **mem**).

La convenzione **C** per il passaggio degli argomenti alle funzioni, prende origine dal fatto che il linguaggio **C** prevede anche il caso di funzioni che accettano un numero variabile di argomenti; un caso emblematico e' rappresentato dalla funzione **printf** che fa parte della libreria standard **stdio** del **C**. Consultando l'header file **stdio.h** si nota la presenza di un prototipo del tipo:
`int printf(char *format, ...);`.
Questo prototipo afferma che **printf** richiede come primo argomento, l'indirizzo di una stringa (vettore di **char**) chiamata **stringa di formato**; i tre puntini seguenti indicano che questa funzione accetta una lista variabile di ulteriori argomenti. Scrivendo ad esempio:
`printf("V[%d] = %d\n", i, vector[i]);`,
stiamo chiamando **printf** con tre argomenti che sono: la stringa di formato, il valore di una variabile **i** e il valore di una variabile **vector[i]** (elemento di indice **i** di un vettore **vector**). Se **i** vale **4** e **vector[i]** vale **10**, verra' prodotto l'output:
`V[4] = 10`,
seguito da un **new line** (avanzamento linea).
I progettisti del linguaggio **C** hanno constatato che il modo migliore per gestire questa situazione, consisteva nel passare gli argomenti a partire dall'ultimo (da destra verso sinistra); osserviamo infatti che in questo modo, qualunque sia il numero di argomenti inseriti nello stack, il primo di essi e' sempre quello successivo all'indirizzo di ritorno, il secondo segue il primo, il terzo segue il secondo e cosi' via. Inserendo invece gli argomenti a partire dal primo (da sinistra verso destra), non siamo in grado di stabilire con certezza quale sia la posizione nello stack del primo di essi; di conseguenza non siamo in grado nemmeno di stabilire la posizione nello stack del secondo argomento, del terzo, etc. Supponiamo ad esempio che la precedente chiamata di **printf** sia di tipo

NEAR; all'interno di **printf** otteniamo allora la seguente situazione:

In posizione **BP+4** e' presente l'offset della stringa di formato.
In posizione **BP+6** e' presente il valore **4** (copia del valore di **i**).
In posizione **BP+8** e' presente il valore **10** (copia del valore di **vector[i]**).

Tutti i concetti esposti in questo paragrafo, verranno approfonditi in un apposito capitolo dedicato all'interfacciamento tra **C** e Assembly.

Convenzione Pascal per il passaggio di argomenti ai sottoprogrammi.

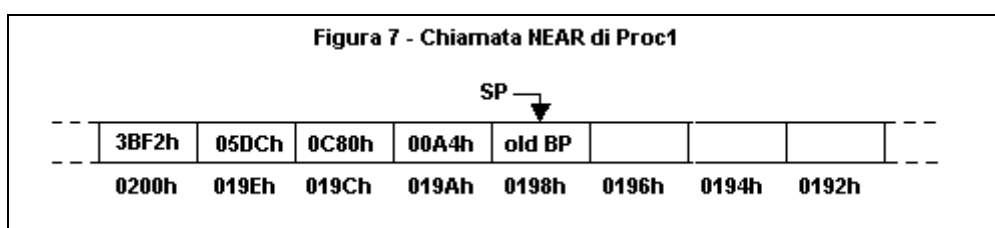
Nel caso del linguaggio **Pascal**, i sottoprogrammi vengono supportati attraverso le due direttive **function** e **procedure**; la **function** implementa il concetto di estensione degli operatori matematici, mentre la **procedure** implementa il concetto di estensione delle istruzioni semplici.
Analizziamo il caso di un generico sottoprogramma **Pascal** visto che le considerazioni che verranno svolte valgono sia per le **function** che per le **procedure**; la convenzione **Pascal** stabilisce che gli argomenti da passare ad un sottoprogramma, debbano essere inseriti nello stack a partire da quello piu' a sinistra (primo argomento). Per illustrare questa convenzione, facciamo riferimento al seguente programma **Pascal**:

```
program Test;

{ variabili globali }

var
   var1, var2: Integer;

{ definizione dei sottoprogrammi }

procedure Proc1(i1, i2: Integer);
begin
   { codice di Proc1 }
end;

{ entry point }

begin
   var1 := 1500;
   var2 := 3200;

   Proc1(var1, var2);
end.
```

Il programma inizia con la definizione di due variabili di tipo **Integer**, chiamate **var1** e **var2**; il tipo **Integer** del **Pascal** e' perfettamente equivalente al tipo **int** del **C**. Tutte le variabili definite all'inizio di un programma **Pascal**, corrispondono alle variabili globali del **C** e quindi vengono inserite dal compilatore **Pascal** nel segmento dati del programma; queste variabili quindi sono visibili e accessibili da qualunque funzione o procedura del programma.
Subito dopo le variabili globali, incontriamo la definizione di una procedura chiamata **Proc1**; come si puo' notare, questa procedura richiede due argomenti di tipo **Integer**.
A questo punto arriviamo all'istruzione **BEGIN** che delimita l'entry point di un programma **Pascal**; all'interno del blocco codice principale del programma e' presente l'inizializzazione di **var1** e **var2**, e la chiamata della procedura **Proc1**.
Anche in questo caso, il nostro obiettivo e' capire come vengono passati gli argomenti **var1** e **var2** a questa procedura. Cominciamo dal caso in cui la procedura **Proc1** sia di tipo **NEAR**, e la sua

definizione si trovi nello stesso segmento di codice del caller; in questo caso il compilatore incontrando la chiamata di **Proc1**, produce il codice macchina corrispondente alle seguenti istruzioni Assembly:

```
push     var1
push     var2
call     near ptr Proc1
```

Rispetto all'esempio della **NEAR** call della funzione **C func1** che abbiamo analizzato in precedenza, si notano due importanti differenze; osserviamo infatti che gli argomenti vengono passati a partire dal primo (da sinistra verso destra), e inoltre, non e' piu' presente l'istruzione:
`add sp, 4.`
Per analizzare questa situazione, ci serviamo come al solito di una rappresentazione visiva dello stato del segmento di stack del programma; prima della chiamata di **Proc1** la situazione dello stack e' identica a quella illustrata in Figura 5a. Subito dopo la chiamata di **Proc1** e il salvataggio del contenuto originale di **BP** otteniamo la situazione descritta nella seguente Figura 7:

In pratica, partendo da **SP=0200h**, la copia del valore **05DCh** di **var1** viene inserita nello stack a **SP=019Eh**, mentre la copia del valore **0C80h** di **var2** viene inserita nello stack a **SP=019Ch**; la chiamata **NEAR** di **Proc1** provoca l'inserimento nello stack dell'indirizzo di ritorno **00A4h** a **SP=019Ah**. All'interno di **Proc1**, l'accesso ai vari parametri avviene come al solito attraverso **BP**; il compilatore **Pascal** inserisce quindi prima di tutto l'istruzione:
`push bp.`
In seguito all'esecuzione di questa istruzione, il contenuto originale (**old BP**) di **BP** viene inserito nello stack a **SP=0198h**; in base a questa situazione, la successiva istruzione:
`mov bp, sp,`
copia in **BP** il valore **0198h** di **SP**.
A questo punto, osservando la Figura 7 possiamo affermare che:

1) L'argomento **05DCh** (copia del valore di **var1**) si trova nello stack all'indirizzo **SS:(BP+6)**; infatti:
BP+6 = 019Eh, e quindi **[BP+6] = 05DCh**.
2) L'argomento **0C80h** (copia del valore di **var2**) si trova nello stack all'indirizzo **SS:(BP+4)**; infatti:
BP+4 = 019Ch, e quindi **[BP+4] = 0C80h**.

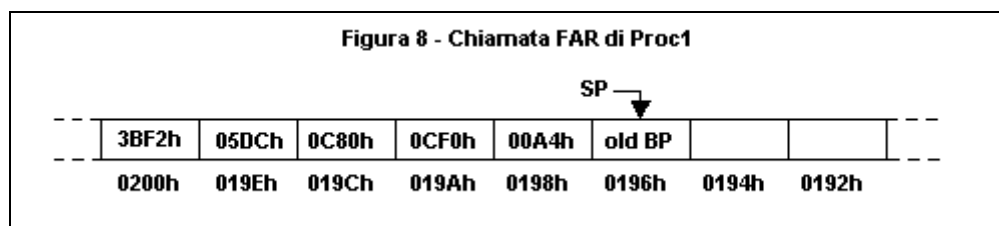
In pratica, l'ultimo argomento inserito nello stack si trova subito dopo l'indirizzo di ritorno, poi incontriamo in successione il penultimo argomento, il terzultimo, etc; il **Pascal** non consente la definizione di sottoprogrammi con lista variabile di argomenti, per cui la posizione nello stack dei vari argomenti e' facilmente individuabile.
Nella parte finale della procedura **Proc1**, il compilatore **Pascal** inserisce le due istruzioni:

```
pop      bp
ret      4
```

La prima di queste istruzioni come al solito ripristina il contenuto originale di **BP**; la seconda istruzione e' un **NEAR** return con operando immediato di valore **4**. Come e' stato spiegato nel Capitolo 19, la CPU incontrando questa istruzione esegue i seguenti passi:

1) Legge il valore **00A4h** dall'indirizzo **SS:SP**, cioe' **SS:019Ah**.
2) Somma 2 byte a **SP** per recuperare lo spazio appena liberatosi nello stack, e ottiene **SP=019Ch**.
3) Somma il valore immediato **4** a **SP** ottenendo **SP=0200h**.
4) Salva in **IP** il valore **00A4h**.
5) Trasferisce il controllo (salta) a **CS:IP**.

Attraverso questo salto, il controllo viene restituito al caller; quando il caller riottiene il controllo, trova il segmento di stack perfettamente ripristinato (Figura 5a).
Come si puo' facilmente intuire, l'operando immediato **4** di **RET** ha il compito di ripulire lo stack dagli argomenti che avevamo passato a **Proc1**; siccome avevamo inserito due word, pari a **4** byte, sommando **4** a **SP=019Ch** si ottiene **SP=0200h**.
In base alla convenzione del linguaggio **Pascal**, il compito di ripulire lo stack dagli argomenti ricade sul called, cioe' sul sottoprogramma che e' stato chiamato; possiamo dire quindi che in relazione al passaggio degli argomenti e alla pulizia dello stack, le convenzioni **Pascal** sono opposte a quelle del **C**.

A questo punto, il caso della **FAR** call di **Proc1** appare abbastanza semplice in quanto il meccanismo dovrebbe essere ormai chiaro; la situazione all'interno di **Proc1** viene descritta dalla Figura 8:

Osservando la Figura 8 possiamo affermare che:

1) L'argomento **05DCh** (copia del valore di **var1**) si trova nello stack all'indirizzo **SS:(BP+8)**; infatti:
BP+8 = 019Eh, e quindi **[BP+8] = 05DCh**.
2) L'argomento **0C80h** (copia del valore di **var2**) si trova nello stack all'indirizzo **SS:(BP+6)**; infatti:
BP+6 = 019Ch, e quindi **[BP+6] = 0C80h**.

Al termine di **Proc1** viene ripristinato il contenuto originale di **BP**, e viene effettuato un **FAR** return con operando immediato di valore **4**; la tecnica utilizzata dal **Pascal** per la pulizia dello stack, e' piu' veloce dell'analoga tecnica utilizzata dal **C**.
Anche il **Pascal** permette il passaggio degli argomenti per indirizzo; a tale proposito, e' necessario utilizzare la parola riservata **VAR**. Supponiamo che la procedura **Proc1** dell'esempio precedente richieda il primo argomento per valore e il secondo per indirizzo; in questo caso, la definizione di **Proc1** assume il seguente aspetto:

```
procedure Proc1(i1: Integer; var i2: Integer)
begin
   { codice di Proc1 }
end;
```

Nel linguaggio **C**, il programmatore ha il compito di passare esplicitamente un argomento per indirizzo; nel caso del **Pascal** invece, tutto il lavoro necessario viene svolto dal compilatore. La chiamata di **Proc1** quindi rimane immutata:

```
Proc1(var1, var2);.
```

In presenza di questa chiamata, il compilatore **Pascal** tenendo conto della definizione di **Proc1**, produce il codice macchina corrispondente alle seguenti istruzioni Assembly (nel caso di **NEAR** call):

```
push     var1
push     offset var2
call     near ptr Proc1
```

La procedura **Proc1** riceve quindi una copia del valore di **var1** e una copia dell'indirizzo di **var2**; accedendo a questo indirizzo, **Proc1** ha la possibilita' di modificare il contenuto originale della variabile **var2**.
Tutti i concetti esposti in questo paragrafo, verranno approfonditi in un apposito capitolo dedicato all'interfacciamento tra **Pascal** e Assembly.

Convenzioni per le variabili locali dei sottoprogrammi.

Come abbiamo avuto modo di constatare nei precedenti capitoli, ogni programma scritto in un qualsiasi linguaggio, ha bisogno di una serie di variabili destinate a contenere dei valori; questi valori possono rappresentare ad esempio indirizzi di memoria, risultati intermedi di calcoli che il programma sta svolgendo, etc. Le variabili di un programma hanno due caratteristiche importantissime che vengono chiamate **durata** e **visibilita'**; nella fase di esecuzione di un programma, la durata indica il ciclo di vita di una variabile, cioe' in quali momenti la variabile esiste (ed e' quindi accessibile) e in quali momenti invece non esiste (e non e' quindi accessibile). La visibilita' indica in quali aree del programma una variabile e' "visibile" (ed e' quindi accessibile) e in quali aree invece non e' visibile (e non e' quindi accessibile); si puo' notare quindi che sia la durata sia la visibilita' hanno a che fare con l'accessibilita' di una variabile.
Puo' capitare che un programma abbia bisogno di variabili che esistano in qualunque momento della fase di esecuzione, e che siano visibili da qualunque punto del programma stesso; questo tipo di variabili vengono denominate **variabili globali**. Come gia' sappiamo, le variabili per poter avere queste caratteristiche, devono essere definite all'interno dei segmenti di programma; preferibilmente e' opportuno inserire queste definizioni nei segmenti di dati. Le variabili definite all'interno di un segmento di programma vengono anche chiamate **variabili statiche** in quanto a ciscuna di esse viene assegnato un indirizzo di memoria che rimane fisso (statico) per tutta la durata del programma; appare evidente quindi che la durata di una variabile globale coincide con la durata dell'intero programma. Una variabile globale ha anche una visibilita' globale in quanto puo' essere vista e quindi acceduta da qualunque punto del programma; in un prossimo capitolo vedremo inoltre che in un programma formato da piu' files, una variabile globale definita in un file puo' essere resa visibile e quindi accessibile agli altri files.
Anche un sottoprogramma puo' avere bisogno di apposite variabili necessarie per poter memorizzare temporaneamente determinate informazioni; in questo caso, l'utilizzo delle variabili globali non rappresenta una soluzione molto conveniente in quanto produce un grave spreco di memoria. Osserviamo infatti che queste variabili vengono utilizzate solo nel momento in cui avviene la chiamata del sottoprogramma; non appena il sottoprogramma termina, le variabili ad esso associate non servono piu' e continuano ad occupare inutilmente la memoria. L'ideale sarebbe "creare" queste variabili al momento della chiamata di un sottoprogramma, e "distruggerle" non appena il sottoprogramma restituisce il controllo al caller; questa e' proprio la strada seguita dai compilatori dei linguaggi di alto livello. Naturalmente, il luogo piu' adatto per creare e distruggere

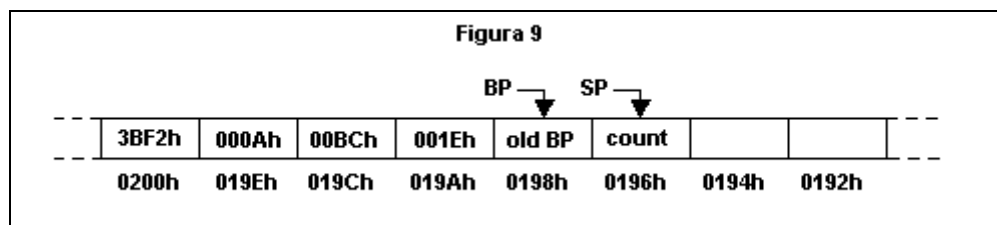
variabili non puo' che essere lo stack; il procedimento che si segue e' analogo a quello che ci permette di passare gli argomenti ai sottoprogrammi. Abbiamo visto infatti che in fase di chiamata di un sottoprogramma, gli argomenti richiesti vengono inseriti nello stack; quando il sottoprogramma termina, lo stack viene ripulito in modo da recuperare la memoria non piu' necessaria.
Nella terminologia dei linguaggi di alto livello, le variabili create all'interno di un sottoprogramma vengono definite **variabili locali**; in contrapposizione alle variabili statiche create nei segmenti di programma, le variabili locali create nello stack possono essere definite **variabili dinamiche**. Nel linguaggio **C** le variabili locali vengono anche chiamate **variabili automatiche**; questa definizione e' legata chiaramente al fatto che queste variabili vengono automaticamente create e distrutte dai sottoprogrammi. La durata di una variabile locale coincide con la durata del sottoprogramma di appartenenza; una variabile locale infatti "nasce" al momento della chiamata del sottoprogramma, e "muore" quando lo stesso sottoprogramma termina. Una variabile locale ha anche visibilita' locale; e' chiaro infatti che solo il sottoprogramma che ha creato una variabile locale, conosce l'offset di questa variabile all'interno dello stack. In base alle considerazioni appena esposte, possiamo dire che anche gli argomenti passati ad un sottoprogramma possono essere considerati a tutti gli effetti come variabili locali.

Vediamo un esempio pratico che illustra il procedimento che permette ai compilatori dei linguaggi di alto livello di creare variabili locali; supponiamo di voler scrivere una procedura che inizializza con il valore zero tutti gli **n** elementi di un qualsiasi vettore di word. Supponiamo poi di avere un vettore di **10** elementi, chiamato **vector1**, e definito all'offset **00BCh** di un blocco dati di nome **DATASEGM**; la procedura richiede due argomenti che rappresentano l'offset iniziale del vettore, e il numero di elementi. All'interno della procedura e' presente un loop che inizializza con zero tutti gli elementi del vettore; in questi casi e' preferibile utilizzare ovviamente **CX** come contatore. Nel nostro esempio invece, a scopo didattico utilizziamo una variabile locale a **16** bit creata nello stack; per il passaggio degli argomenti e per la pulizia dello stack seguiamo inoltre la convenzione **C**. La struttura della procedura di tipo **NEAR** chiamata **ZeroVector** e' la seguente:

```
; void ZeroVector(int *voffset, int n)

ZeroVector proc near

   push     bp                   ; preserva il vecchio bp
   mov      bp, sp               ; ss:bp = ss:sp
   sub      sp, 2                ; creazione variabili locali

   mov      bx, [bp+4]           ; bx = voffset
   mov      ax, [bp+6]           ; ax = n
   mov      [bp-2], ax           ; contatore = n
   xor      ax, ax               ; ax = 0
zero_loop:
   mov      [bx], ax             ; azzera l'elemento corrente
   add      bx, 2                ; punta al prossimo elemento
   dec      word ptr [bp-2]      ; decremento contatore
   jnz      short zero_loop      ; controllo loop

   mov      sp, bp               ; distruzione variabili locali
   pop      bp                   ; ripristino vecchio bp
   ret                           ; near return

ZeroVector endp
```

Il parametro **voffset** rappresenta l'indirizzo di un **int** che nel nostro caso e' l'indirizzo del primo elemento di **vector1**; il parametro **n** rappresenta il numero di elementi del vettore. All'interno di **ZeroLoop**, l'accesso agli argomenti avviene nel solito modo; osserviamo che trattandosi di una procedura **NEAR**, l'argomento **voffset** si trova a **[BP+4]**, mentre l'argomento **n** si trova a **[BP+6]**.
La novita' e' rappresentata dall'istruzione:
`sub sp, 2.`
Sottraendo **2** a **SP**, stiamo richiedendo allo stack **2** byte di spazio da destinare ad una variabile locale che utilizzeremo come contatore; questa e' un'altra rarissima circostanza nella quale il programmatore e' chiamato a gestire direttamente lo stack pointer **SP**. Come si puo' notare, il contatore che si trova quindi a **[BP-2]**, viene inizializzato attraverso **AX** con il valore **n**. All'interno del loop, il contatore viene continuamente decrementato finche' non diventa zero; questa e' la condizione di uscita dal loop. Al termine del loop, e' presente l'istruzione:
`mov sp, bp.`
Questa istruzione ripulisce lo stack dalle variabili locali in quanto riposiziona **SP** all'offset che aveva prima della creazione del contatore; potevamo anche scrivere:
`add sp, 2.`
Infine incontriamo la solita istruzione di ripristino di **BP** e il **NEAR** return che restituisce il controllo al caller.
La chiamata **C** di **ZeroVector** e' la seguente:

```
push     word ptr 10       ; passa n
push     offset vector1    ; passa voffset
call     ZeroVector        ; chiama la procedura
add      sp, 4             ; ripulisce lo stack
```

La Figura 9 chiarisce meglio la situazione che si viene a creare all'interno dello stack in seguito alla chiamata di **ZeroVector** e alla creazione della variabile locale.

Come al solito, supponiamo di partire da **SP=0200h** che punta alla word **3BF2h** gia' presente nello stack; a questo punto inizia il passaggio degli argomenti alla procedura. L'argomento **n = 10 = 000Ah** (numero di elementi del vettore) viene inserito a **SP=019Eh**; l'argomento **offset vector1 = 00BCh** viene inserito a **SP=019Ch**. La chiamata **NEAR** di **ZeroVector** determina l'inserimento nello stack del solo offset dell'indirizzo di ritorno; nel nostro esempio quest'offset vale **001Eh** e viene inserito a **SP=019Ah**. All'interno di **ZeroVector** viene prima di tutto salvato il contenuto originario **old BP** di **BP** nello stack; il valore **old BP** viene inserito a **SP=0198h**. In base a questa situazione, l'istruzione:
`mov bp, sp,`
copia in **BP** il valore **0198h** di **SP**.

A questo punto avviene la creazione nello stack di una variabile locale a **16** bit indicata in Figura 9 con il nome **count**; a tale proposito sottraiamo **2** byte a **SP** ottenendo **SP=0196h**, mentre **BP** vale sempre **0198h**. Tenendo conto dei valori di **SP** e **BP** possiamo dire quindi che:

1) L'argomento **voffset = 00BCh** si trova a **BP+4**.
2) L'argomento **n = 000Ah** si trova a **BP+6**.
3) La variabile locale **count** si trova a **BP-2**.
In sostanza, gli argomenti della procedura si trovano a spiazzamenti positivi da **BP**; invece le variabili locali si trovano a spiazzamenti negativi da **BP**.
Nella parte finale della procedura incontriamo le istruzioni di ripristino; la prima di queste istruzioni e':
`mov sp, bp.`
Questa istruzione copia in **SP** il contenuto **0198h** di **BP**; in questo modo lo stack viene ripulito dalle variabili locali create dalla procedura; l'istruzione seguente ripristina **BP** e pone **SP=019Ah**. Il **NEAR** return estrae **001Eh** dallo stack, lo carica in **IP**, pone **SP=019Ch** e salta a **CS:IP**; a **CS:IP** (cioe' a **CS:001Eh**) viene incontrata l'istruzione:
`add sp, 4,`
che ripulisce lo stack dagli argomenti passati a **ZeroVector**; a questo punto otteniamo **SP=0200h** con il segmento di stack completamente ripristinato.

Nei linguaggi di alto livello, l'inserimento degli argomenti nello stack, e la sequenza di istruzioni:

```
push     bp
mov      bp, sp
sub      sp, 2
```

rappresentano il **prolog code** (codice di prologo) o **procedure entry**; invece la sequenza di istruzioni:

```
mov      sp, bp
pop      bp
```

piu' l'istruzione per ripulire lo stack dai parametri, rappresentano l'**epilog code** (codice di epilogo) o **procedure exit**. All'interno di una procedura, ulteriori istruzioni di inserimento e di estrazione che coinvolgono lo stack, devono trovarsi tassativamente dopo il prolog code e prima dell'epilog code; in caso contrario tutto lo schema di Figura 9 va a farsi benedire e il disastro e' assicurato.

Supponiamo ad esempio di voler preservare il contenuto originale di tutti i registri utilizzati da **ZeroVector**; in questo caso dobbiamo scrivere:

```
; void ZeroVector(int *voffset, int n)

ZeroVector proc near

   push     bp                   ; preserva il vecchio bp
   mov      bp, sp               ; ss:bp = ss:sp
   sub      sp, 2                ; creazione variabili locali
   push     ax                   ; preserva il vecchio ax
   push     bx                   ; preserva il vecchio bx

   mov      bx, [bp+4]           ; bx = voffset
   mov      ax, [bp+6]           ; ax = n
   mov      [bp-2], ax           ; contatore = n
   xor      ax, ax               ; ax = 0
zero_loop:
   mov      [bx], ax             ; azzera l'elemento corrente
   add      bx, 2                ; punta al prossimo elemento
   dec      word ptr [bp-2]      ; decremento contatore
   jnz      short zero_loop      ; controllo loop

   pop      bx                   ; ripristino vecchio bx
   pop      ax                   ; ripristino vecchio ax
   mov      sp, bp               ; distruzione variabili locali
   pop      bp                   ; ripristino vecchio bp
   ret                           ; near return

ZeroVector endp
```

Come al solito, e' importantissimo ricordare che tutte le estrazioni dallo stack devono avvenire in senso inverso rispetto agli inserimenti.

Le convenzioni del **Pascal** per la creazione delle variabili locali, sono identiche alle convenzioni del **C**; anche nel caso del **Pascal** quindi, all'interno di un sottoprogramma gli argomenti si trovano a spiazzamenti positivi da **BP**, mentre le variabili locali si trovano a spiazzamenti negativi da **BP**. In definitiva, nei linguaggi di alto livello, tutta la gestione degli argomenti e delle variabili locali ruota attorno al base pointer **BP**; questi linguaggi richiedono tassativamente che all'interno dei sottoprogrammi che fanno uso di argomenti e di variabili locali, il contenuto originale di **BP** debba essere sempre preservato.
Tutti questi concetti vengono approfonditi in modo dettagliato nei capitoli dedicati all'interfaccia tra l'Assembly e i linguaggi di alto livello.

Gestione degli argomenti e delle variabili locali con EQU.

All'interno di una procedura, la gestione diretta degli argomenti e delle variabili locali attraverso **BP** risulta particolarmente scomoda; il programmatore deve ricordarsi di calcolare il corretto spiazzamento ogni volta che vuole accedere ad un determinato argomento o ad una determinata variabile locale. Questa situazione determina una serio incremento delle possibilita' di commettere errori; per ovviare a questo inconveniente, e' possibile utilizzare la direttiva **EQU**. Come sappiamo, questa direttiva e' in grado di gestire attraverso un nome simbolico anche stringhe alfanumeriche; l'importante e' che queste stringhe, una volta espanse, producano codice Assembly sintatticamente e semanticamente valido. Utilizzando allora la direttiva **EQU**, la procedura **ZeroVector** puo' essere riscritta in questo modo:

```
; void ZeroVector(int *voffset, int n)

ZeroVector proc near

   voffset  EQU   [bp+4]          ; indirizzo iniziale vettore
   n        EQU   [bp+6]          ; numero elementi del vettore
   count    EQU   [bp-2]          ; contatore

   push     bp                    ; preserva il vecchio bp
   mov      bp, sp                ; ss:bp = ss:sp
   sub      sp, 2                 ; creazione variabili locali

   mov      bx, voffset           ; bx = voffset
   mov      ax, n                 ; ax = n
   mov      count, ax             ; contatore = n
   xor      ax, ax                ; ax = 0
zero_loop:
   mov      [bx], ax              ; azzera l'elemento corrente
   add      bx, 2                 ; punta al prossimo elemento
   dec      word ptr count        ; decremento contatore
   jnz      short zero_loop       ; controllo loop

   mov      sp, bp                ; distruzione variabili locali
   pop      bp                    ; ripristino vecchio bp
   ret                            ; near return

ZeroVector endp
```

Come si puo' notare, una volta dichiarate le varie macro, la gestione degli argomenti e delle variabili locali diventa semplicissima; inoltre, una procedura scritta in questo modo appare molto piu' elegante e comprensibile.
Quando si dichiarano nomi simbolici per le macro, bisogna stare attenti a non utilizzare nomi gia' usati in precedenza per definire variabili, etichette, procedure, etc; e' possibile invece ridichiarare nomi gia' associati con **EQU** ad altre stringhe alfanumeriche. La visibilita' dei nomi delle macro e' limitata al solo file di appartenenza.
Un'ultima considerazione riguarda il fatto che nella fase di progettazione di una procedura che utilizza argomenti e variabili locali, e' importantissimo tracciare sempre su un foglio di carta, uno schema dello stack; un disegno come quello di Figura 9 rappresenta un notevole aiuto che riduce enormemente il rischio di commettere errori. Come offset iniziale si puo' utilizzare un qualunque valore simbolico come ad esempio **SP=0400h**; si ricorda inoltre che e' conveniente tracciare lo schema dello stack con indirizzi crescenti da destra verso sinistra, dall'alto verso il basso o dal basso verso l'alto, ma mai da sinistra verso destra.

L'indirizzo di una variabile locale.

In relazione alle variabili locali, e' necessario affrontare un aspetto delicatissimo di cui il programmatore Assembly deve sempre tenere presente; la non conoscenza di questo aspetto puo' portare alla scrittura di programmi contenenti pericolosissimi bugs.
Abbiamo visto che le variabili statiche sono cosi' chiamate in quanto, essendo definite all'interno di un segmento di programma, sono dotate di un ben preciso indirizzo che rimane fisso (statico) per tutta la fase di esecuzione del programma; in fase di assemblaggio del programma, l'assembler ogni volta che incontra la definizione di una nuova variabile, assegna ad essa un offset che rappresenta la distanza in byte che separa la definizione della variabile dall'inizio del segmento di appartenenza.
Supponiamo di avere una variabile a **16** bit, chiamata **Var16**, e definita all'offset **0010h** di un segmento dati **DATASEGM**; possiamo dire quindi che la definizione di **Var16** si trova a **0010h** byte di distanza dall'inizio del blocco **DATASEGM**. In un caso del genere, e' lecito scrivere istruzioni del tipo:

```
mov bx, offset Var16.
```

Quando l'assembler incontra questa istruzione, genera un codice macchina che, in fase di esecuzione del programma, dice alla CPU di caricare in **BX** il valore immediato **0010h**; tutto cio' e' possibile in quanto l'offset di **Var16** all'interno di **DATASEGM**, e' un valore immediato che l'assembler e' in grado di determinare in modo univoco. Le considerazioni appena esposte, si applicano in generale al caso in cui **Var16** venga definita staticamente all'interno di qualsiasi segmento di programma (codice, dati o stack); se ad esempio definiamo staticamente **Var16** all'interno del segmento di stack **STACKSEGM**, l'assembler incontrando l'istruzione precedente, calcola l'offset di **Var16** rispetto a **STACKSEGM**.
La situazione cambia radicalmente nel caso delle variabili locali (compresi gli argomenti passati ad una procedura); queste variabili non possono avere un indirizzo fisso (statico), in quanto esse vengono create e distrutte dinamicamente nello stack. Consideriamo ad esempio gli argomenti **voffset** e **n**, e la variabile locale **count** della procedura **ZeroVector**; ogni volta che **ZeroVector** viene chiamata, l'unica cosa che possiamo dire e' che **voffset** si trova nello stack a **BP+4**, **n** si trova a **BP+6**, mentre **count** si trova a **BP-2**. L'aspetto fondamentale da sottolineare e' che in fase di assemblaggio del programma, l'assembler non puo' sapere quanto varra' **BP** al momento della chiamata di **ZeroVector**; osserviamo infatti che il contenuto di **BP** dipende dal contenuto di **SP**, e questi valori sono noti solo in fase di esecuzione del programma (run time). Se il programma principale chiama **10** volte **ZeroVector**, puo' capitare che per ciascuna di queste chiamate **SP** assuma un valore differente; questo valore dipende infatti dalla situazione dello stack al momento della chiamata della procedura.
Dalle considerazioni appena esposte, si deduce che nel caso ad esempio della variabile locale **count** di **ZeroVector**, non avrebbe alcun senso scrivere:

```
mov bx, offset count.
```

Purtroppo pero' l'assembler incontrando questa istruzione non visualizza nessun messaggio di errore o di avvertimento; di conseguenza, e' importantissimo che il programmatore ricordi sempre di non utilizzare nella maniera piu' assoluta l'operatore **OFFSET** per determinare l'indirizzo degli argomenti o delle variabili locali di una procedura. Per ovviare a questo problema, esistono diverse soluzioni; volendo caricare ad esempio in **BX** l'offset di **count**, potremmo scrivere:

```
mov bx, bp-2;
```

questa istruzione pero' e' illegale perche' l'assembler (come al solito) non e' in grado di sapere quanto vale **BP** e non e' quindi in grado di valutare l'espressione **BP-2**. Questo lavoro deve essere delegato alla CPU che in fase di esecuzione puo' eseguire i calcoli necessari; in fase di assemblaggio dobbiamo scrivere quindi:

```
mov      bx, bp
sub      bx, 2
```

A questo punto possiamo scrivere anche istruzioni del tipo:

```
mov ax, [bx];
```

questa istruzione carica in **AX** il contenuto di **count**.
Ricordando quanto e' stato detto nel Capitolo 15 a proposito dell'istruzione **LEA**, possiamo anche scrivere:

```
lea bx, count,
```

che equivale ovviamente a scrivere:

```
lea bx, [bp-2].
```

Indubbiamente questo e' il metodo da preferire; osserviamo infatti che con **LEA** possiamo utilizzare direttamente i nomi simbolici come **count**, **voffset**, etc.
E' chiaro che tutte le istruzioni appena illustrate hanno senso solo all'interno di **ZeroVector**; infatti, quando **ZeroVector** termina, le sue variabili locali (compresi gli argomenti ricevuti) vengono distrutte e quindi non esistono piu' fino alla successiva chiamata. Da tutto cio' si deduce anche che e' un gravissimo errore restituire al caller l'indirizzo di una variabile locale; questo concetto dovrebbe essere ben noto ai programmatori **C**.

Convenzioni per i valori di ritorno dei sottoprogrammi.

Molto spesso si presenta la necessita' di scrivere sottoprogrammi che ricevono una lista di argomenti, li elaborano attraverso un apposito algoritmo e producono infine un risultato da restituire al caller; questo valore restituito al caller viene definito **return value** (valore di ritorno). I linguaggi di alto livello utilizzano apposite convenzioni che stabiliscono in quale modo si deve "spedire" il valore di ritorno al caller; fortunatamente, in relazione al return value, i diversi linguaggi di alto livello seguono convenzioni molto simili tra loro. In particolare, i valori interi vengono sempre restituiti attraverso l'accumulatore **AX/EAX** in congiunzione se necessario con **DX/EDX**; i valori in virgola mobile (numeri reali), vengono invece restituiti attraverso **ST(0)** che rappresenta la cima dello stack (**TOS**) del coprocessore matematico (**FPU**). La tabella seguente illustra in quale registro (locazione) vengono restituiti i vari tipi di dati del liguaggio **C** e del **Pascal**; questa tabella e' valida quindi anche per i linguaggi "derivati", come il **C++**, l'**Object Pascal**, **Delphi**, etc.

Locazioni dei valori di ritorno			
Tipo		**Dimensione (bit)**	**locazione**
C	**Pascal**		
char	Shortint	8	AX
unsigned char	Byte	8	AX
short	Integer	16	AX
unsigned short	Word	16	AX
int	Integer	16	AX
unsigned int	Word	16	AX
long	Longint	32	DX:AX
unsigned long	---	32	DX:AX
float	single	32	ST(0)
double	double	64	ST(0)
long double	extended	80	ST(0)
NEAR addr.	NEAR addr.	16	AX
FAR addr.	FAR addr.	32	DX:AX

In pratica, i valori interi a **8** bit vengono restituiti nel byte meno significativo di **AX** (half register **AL**); in questo caso il contenuto di **AH** non e' significativo e dovrebbe valere zero (in ogni caso e' meglio non fidarsi). I valori interi a **16** bit vengono restituiti in **AX**. I valori interi a **32** bit vengono restituiti nella coppia **DX:AX**; nel rispetto della convenzione **little endian**, il registro **DX** contiene la word piu' significativa, mentre **AX** contiene la word meno significativa. Queste considerazioni valgono solo per la modalita' reale a **16** bit del **DOS**; in questa modalita', la dimensione standard dei tipi interi del **C** e' pari a **8** bit per i **char**, **16** bit per gli **short**, **16** bit per gli **int** e **32** bit per i **long**.
Nella modalita' protetta a **32** bit dei **SO** come **Windows**, **Unix/Linux**, **OS/2**, etc, la dimensione standard dei tipi interi e' pari a **8** bit per i **char**, **16** bit per gli **short**, **32** bit per gli **int** e **32** bit per i **long**; in questa modalita', i valori interi a **32** bit vengono restituiti in **EAX**, mentre i valori interi a **64** bit vengono restituiti in **EDX:EAX**.
La convenzione **little endian** viene applicata anche per i valori di ritorno di tipo indirizzo (**address**); gli indirizzi **NEAR** della modalita' reale sono formati da un solo offset a **16** bit e vengono restituiti in **AX**. Gli indirizzi **FAR** della modalita' reale sono formati da una coppia **seg:offset** a **16+16** bit e vengono restituiti in **DX:AX**; il registro **DX** contiene la componente **seg**, mentre **AX** contiene la componente **offset**. Nella modalita' protetta a **32** bit, per indirizzo **NEAR** si intende un offset a **32** bit; questo tipo di indirizzo viene restituito in **EAX**.
Tutti i valori reali in floating point, vengono restituiti nel registro **ST(0)** che rappresenta la cima dello stack (**TOS**) della **FPU**; questo aspetto viene illustrato nella sezione **Assembly Avanzato**.

Programmi Assembly vecchio stile.

Se si analizza il listato di qualche vecchio programma Assembly, si scopre che il blocco delle istruzioni principali si trova usualmente inserito all'interno di una procedura; nel caso di un eseguibile in formato **EXE**, si puo' presentare ad esempio una situazione di questo genere:

```
main proc far             ; entry point

   push     ds            ; salva ds nello stack
   xor      ax, ax        ; ax = 0
   push     ax            ; salva 0 nello stack

   mov      ax, DATASEGM  ; carica DATASEGM
   mov      ds, ax        ; in ds
   assume   ds: DATASEGM  ; assegna DATASEGM a ds

;------ inizio blocco principale istruzioni ------

;------- fine blocco principale istruzioni -------

   ret                    ; far return

main endp
```

Alla fine del file che contiene il programma Assembly, troviamo inoltre la direttiva:
`END main`;
questa direttiva come sappiamo indica che l'identificatore **main** (che in questo caso e' il nome di una procedura) rappresenta l'entry point del programma.
Come si puo' notare, alla fine della procedura **main** non sono presenti le solite istruzioni che attraverso il servizio **4Ch** dell'**INT 21h** terminano il programma restituendo il controllo al **DOS**; vediamo allora di capire come si svolgeva questa fase nei vecchi programmi Assembly.
Come e' stato spiegato nel Capitolo 13, al momento di caricare in memoria un eseguibile in formato **EXE**, il **DOS** crea due blocchi di memoria; nel primo blocco viene sistemato l'**Environment Segment** del programma, mentre nel secondo blocco vengono sistemati i **256** byte del **Program**

Segment Prefix (**PSP**) seguiti dal programma vero e proprio. Subito dopo il **DOS** procede con l'inizializzazione dei vari registri della CPU; nei due registri **DS** e **ES** viene caricato il paragrafo di memoria da cui parte il **PSP**, per cui si ottiene **DS=PSP** e **ES=PSP**. Nel registro **CS** viene caricato il paragrafo di memoria da cui parte il segmento di programma contenente l'entry point, mentre **IP** viene fatto puntare allo stesso entry point; infine, se e' presente un segmento di programma con attributo di combinazione **STACK**, il **DOS** carica in **SS** il paragrafo di memoria da cui parte questo segmento, e posiziona **SP** alla fine del segmento stesso.
Terminata l'inizializzazione dei vari registri, puo' partire l'esecuzione del programma; la prima istruzione che viene eseguita dalla CPU e' ovviamente quella che si trova all'indirizzo logico **CS:IP**. In base alle inizializzazioni precedenti, possiamo dire che a questo indirizzo si trova la prima istruzione della procedura **main**; questo e' il punto esatto (entry point) in cui il programmatore riceve il controllo. Le prime importantissime istruzioni della procedura **main** provvedono ad inserire nello stack due valori a **16** bit; il primo valore e' il contenuto corrente di **DS**, e in base a quanto e' stato detto in precedenza, si tratta del paragrafo di memoria da cui inizia il **PSP** (al posto di **DS** si puo' usare anche **ES**). La seconda word inserita nello stack e' **AX=0000h**; e' fondamentale che queste istruzioni siano le prime in assoluto all'interno della procedura **main**.
Al termine della procedura **main** la CPU incontra un **FAR** return; di conseguenza si verifica l'estrazione dallo stack di due valori a **16** bit che verranno inseriti in **CS:IP**. Se non ci sono stati errori nella gestione dello stack, questi due valori sono proprio quelli salvati all'inizio di **main**, e cioe' **PSP** e **0000h**; la CPU pone quindi **CS:IP = PSP:0000h** e salta a **CS:IP**. Se osserviamo la Figura 6 del Capitolo 13 (struttura del **PSP**), possiamo notare che all'offset **0000h** del **PSP** e' presente una word chiamata **TerminateLocation**; questa word contiene il codice macchina **20CDh** dell'istruzione:

```
INT 20h
```

Per dimostrarlo possiamo servirci della procedura **writeHex16** della libreria **EXELIB**; a tale proposito, all'interno della procedura **main** e subito dopo l'inizializzazione di **DS**, possiamo scrivere:

```
mov      ax, es:[0000h]     ; ax = [PSP:0000h]
call     writeHex16         ; visualizza ax
```

La prima istruzione carica in **AX** la word contenuta all'indirizzo **ES:0000h = PSP:0000h**; la seconda istruzione visualizza in esadecimale il contenuto di **AX**. In questo modo possiamo constatare che sullo schermo viene mostrata proprio la word **20CDh**; questa word e' chiaramente disposta in notazione **little-endian**, per cui **CDh** che e' il codice macchina di **INT**, e' seguito dal valore immediato **20h**.
Il codice macchina **20CDh** rappresenta una istruzione di terminazione dei programmi **DOS**; come e' stato spiegato nel Capitolo 13, si tratta di un vecchio sistema che si utilizzava nel Sistema Operativo **CP/M** che e' l'antenato del **DOS**. Questa tecnica e' ancora utilizzabile per motivi di compatibilita' con i vecchi programmi **DOS**; in ogni caso, si consiglia vivamente di utilizzare sempre il servizio **4Ch** dell'**INT 21h** che e' in grado di effettuare una terminazione molto piu' sofisticata dei programmi.
E' chiaro che un programmatore Assembly sufficientemente smaliziato, gestirebbe tutta questa situazione anche in altri modi; l'aspetto fondamentale da ricordare e' che alla fine del nostro programma dobbiamo costringere la CPU a saltare a **CS:IP = PSP:0000h**. Un primo metodo alternativo che possiamo seguire, consiste nell'eliminare la procedura **main** sfruttando solamente il **FAR** return; osserviamo infatti che lo scopo della procedura **main** di tipo **FAR** e' solo quello di costringere l'assembler a generare il codice macchina di un **FAR** return. Possiamo ottenere lo stesso risultato inserendo direttamente una istruzione come **RETF** (o il suo codice macchina **CBh**); utilizzando ad esempio l'etichetta **main** come entry point ordinario, possiamo scrivere:

```
main                    ; entry point

   push     ds             ; salva ds nello stack
   xor      ax, ax         ; ax = 0
   push     ax             ; salva 0 nello stack

   mov      ax, DATASEGM   ; carica DATASEGM
   mov      ds, ax         ; in ds
   assume   ds: DATASEGM   ; assegna DATASEGM a ds

;------ inizio blocco principale istruzioni ------

;------- fine blocco principale istruzioni -------

   retf                    ; far return
```

Quando la CPU incontra il codice macchina (**CBh**) di **RETF**, estrae due word dallo stack, le carica in **CS:IP** e salta a **CS:IP**; in assenza di errori nella gestione dello stack, queste due word sono proprio **PSP** e **0000h**.
Un'altro metodo alternativo consiste nel definire una variabile a **32** bit del tipo:
`TermAddress dd 0`;
in questa dword possiamo caricare la coppia **PSP:0000h** che ci serve per terminare il nostro programma. Come al solito, bisogna ricordare che la componente **offset** deve occupare i **16** bit meno significativi della dword; nel momento in cui vogliamo terminare il programma, non dobbiamo fare altro che saltare con **JMP** all'indirizzo contenuto in **TermAddress** (salto indiretto intersegmento). Il codice che si ottiene e' il seguente:

```
main                                        ; entry point

   mov      ax, DATASEGM                    ; carica DATASEGM
   mov      ds, ax                          ; in ds
   assume   ds: DATASEGM                    ; assegna DATASEGM a ds

   mov      word ptr TermAddress[0], 0     ; 0000h
   mov      word ptr TermAddress[2], es    ; PSP

;------ inizio blocco principale istruzioni ------

;------- fine blocco principale istruzioni -------

   jmp      dword ptr TermAddress          ; salta a PSP:0000h
```

Osserviamo che in **TermAddress[2]** viene caricato **ES** in quanto **DS** e' stato modificato (**DS=DATASEGM**); l'inizializzazione di **DS** presuppone che **TermAddress** sia stata definita nel blocco **DATASEGM**.

Passiamo ora al caso dei programmi eseguibili in formato **COM** (COre iMage); in questo caso si puo' presentare ad esempio una situazione di questo genere:

```
COMSEGM  SEGMENT PARA PUBLIC USE16 'SEG_UNICO'

   assume   cs: COMSEGM, ds: COMSEGM
   assume   es: COMSEGM, ss: COMSEGM

   org      0100h          ; 256 byte per il PSP

main proc near             ; entry point

;------ inizio blocco principale istruzioni ------

;------- fine blocco principale istruzioni -------

   ret                     ; near return

main endp

;------------ inizio blocco procedure ------------

;------------- fine blocco procedure -------------

;--------------- inizio blocco dati --------------

;---------------- fine blocco dati ---------------

COMSEGM  ENDS

   END      main
```

Come al solito, per capire il perche' di questa struttura, dobbiamo analizzare il procedimento seguito dal **DOS** per caricare un programma in formato **COM** in memoria; anche in questo caso (vedere il Capitolo 13), al momento di caricare il programma in memoria, il **DOS** rende disponibili due blocchi di memoria. Il primo blocco e' riservato all'**Environment Segment** del programma; il secondo blocco e' formato da **65536** byte ed e' destinato a contenere i **256** byte del **PSP** seguiti dal programma vero e proprio. Successivamente il **DOS** provvede ad inizializzare i vari registri della CPU; nel caso del nostro esempio il **DOS** pone **CS=DS=ES=SS=COMSEGM** (con **COMSEGM=PSP**). Il registro **IP** viene fatto puntare all'entry point **main** (**IP=0100h**); il registro **SP** viene posizionato all'ultimo offset pari (**65534**) del blocco di memoria da **65536** byte. L'ultimo passo compiuto dal **DOS** consiste nell'inserimento nello stack del valore a **16** bit **0000h**; questo passo viene eseguito esclusivamente per i programmi eseguibili in formato **COM**.
Terminata l'inizializzazione dei vari registri, puo' partire l'esecuzione del programma; la prima istruzione che viene eseguita dalla CPU e' ovviamente quella che si trova all'indirizzo logico **CS:IP**. In base alle inizializzazioni precedenti, possiamo dire che a questo indirizzo si trova la prima istruzione della procedura **main**; questo e' il punto esatto (entry point) in cui il programmatore riceve il controllo.
Al termine della procedura **main** (di tipo **NEAR**) la CPU incontra un **NEAR** return; di conseguenza si verifica l'estrazione dallo stack di un valore a **16** bit che verra' caricato in **IP**. Se non ci sono stati errori nella gestione dello stack, questo valore e' proprio quello salvato inizialmente dal **DOS**, e cioe' **0000h**; la CPU pone quindi **IP=0000h** e salta a **CS:IP**. Trattandosi di un programma eseguibile in formato **COM**, il registro **CS** non viene modificato in quanto contiene gia' il paragrafo di memoria da cui inizia il **PSP** (cioe' **CS=PSP**); in base alle considerazioni appena svolte, possiamo notare che anche in questo caso al termine di **main** si ottiene **CS:IP = PSP:0000h**. La CPU esegue quindi l'istruzione **20CDh** con conseguente terminazione del programma e restituzione

del controllo al **DOS**; anche nel caso dei programmi **COM** si possono utilizzare i metodi alternativi descritti in precedenza, ricordando che questa volta dobbiamo servirci di un **NEAR** return o di un salto indiretto intrasegmento.

Capitolo 26 - La ricorsione

Un determinato procedimento si definisce **ricorsivo** quando contiene un richiamo a se stesso; la ricorsione puo' essere diretta o indiretta. Si parla di **ricorsione diretta** o **auto ricorsione** quando un procedimento richiama direttamente se stesso; si parla invece di **ricorsione indiretta** o **mutua ricorsione** quando un procedimento **A** richiama un procedimento **B** che a sua volta richiama di nuovo il procedimento **A**. In generale, la ricorsione indiretta puo' coinvolgere **n** procedimenti **A1, A2, A3, ..., An**; in questo caso, il procedimento **A1** richiama il procedimento **A2** che richiama il procedimento **A3**, e cosi' via, sino al procedimento **An** che richiama di nuovo il procedimento **A1**.

In matematica troviamo numerosi esempi di procedimenti ricorsivi; in questo caso, il termine procedimento e' riferito ad un algoritmo identificato da una funzione matematica. Come gia' sappiamo, una funzione matematica e' dotata di una lista di argomenti (variabili indipendenti) che vengono elaborati da un apposito algoritmo in modo da ottenere un risultato finale (variabile dipendente); possiamo dire quindi che un algoritmo e' ricorsivo quando la sua definizione contiene un richiamo (diretto o indiretto) all'algoritmo stesso.
Consideriamo ad esempio il caso della funzione **fattoriale**; dato un numero intero positivo **n**, si definisce fattoriale di **n** il prodotto dei primi **n** numeri interi positivi consecutivi decrescenti, a partire da **n**. Simbolicamente, il fattoriale di **n** si scrive **n!**; inoltre, per convenzione si pone:
`0! = 1` (per la dimostrazione si puo' consultare un testo di matematica).
Se ad esempio, **n=5**, si ottiene:
`5! = 5 * 4 * 3 * 2 * 1 = 120.`
Si osserva subito che:
`5! = 5 * 4 * 3 * 2 * 1 = 5 * (4 * 3 * 2 * 1) = 5 * 4! = 5 * (5 - 1)!.`
In sostanza, si puo' scrivere la relazione generale:
`n! = n * (n - 1)!.`
Questa relazione dice che il fattoriale di **n** si ottiene moltiplicando **n** per il fattoriale di **n-1**; l'algoritmo appena enunciato e' chiaramente ricorsivo in quanto contiene un riferimento a se stesso.
Un algoritmo ricorsivo puo' essere paragonato ad una sorta di loop dal quale si esce non appena viene raggiunta una determinata condizione (condizione di uscita); nel caso del fattoriale di **n** si nota subito che la condizione di uscita e' rappresentata da:
`(n - 1) = 0`, e cioe': `n = 1.`
In questo caso infatti si ha:
`(n - 1)! = (1 - 1)! = 0! = 1.`
Vediamo allora come si applicano questi concetti per calcolare ricorsivamente il fattoriale del numero **5**; possiamo scrivere:

```
5! = 5 * (5 - 1)! = 5 * 4!
4! = 4 * (4 - 1)! = 4 * 3!
3! = 3 * (3 - 1)! = 3 * 2!
2! = 2 * (2 - 1)! = 2 * 1!
1! = 1 * (1 - 1)! = 1 * 0!
0! = 1
```

A questo punto abbiamo ottenuto tutti i fattori necessari, per cui, procedendo a ritroso si ottiene:
`1 * 1 * 2 * 3 * 4 * 5 = 120.`

Un'altro caso di funzione matematica definibile ricorsivamente, e' rappresentato dall'elevamento a **potenza** intera positiva, di un numero intero positivo diverso da **1**; dati due numeri interi positivi **m** e **n**, con **n** diverso da **1**, si definisce potenza di ordine **m** del numero **n**, il prodotto di **m** fattori tutti uguali a **n**. Simbolicamente, la potenza di ordine **m** del numero **n** viene rappresentata come $\mathbf{n^m}$; se

ad esempio, **n=2** e **m=4** possiamo scrivere quindi:

$2^4 = 2 * 2 * 2 * 2 = 16$.

Anche in questo caso, si osserva subito che:

$2^4 = 2 * 2 * 2 * 2 = 2 * (2 * 2 * 2) = 2 * 2^3 = 2 * 2^{4 - 1}$.

Possiamo scrivere quindi la relazione generale:

$n^m = n * n^{m - 1}$.

Questa relazione dice che la potenza di ordine **m** del numero **n** si ottiene moltiplicando **n** per la potenza di ordine **m-1** del numero **n**; anche questo algoritmo e' chiaramente ricorsivo in quanto contiene un riferimento a se stesso.
Per calcolare questa funzione in modo ricorsivo, dobbiamo determinare la condizione di uscita dalla ricorsione; questa condizione puo' essere facilmente individuata osservando che ad esempio si ha:

$$\frac{2^5}{2^3} = \frac{2 * 2 * 2 * 2 * 2}{2 * 2 * 2} = 2 * 2 = 2^2 = 2^{5 - 3}$$

Dati allora tre numeri interi positivi **n**, **a** e **b**, con **n** diverso da **1**, possiamo scrivere quindi in generale:

$$\frac{n^a}{n^b} = n^{a - b}$$

Da questa formula ricaviamo subito:

$$n^0 = n^{m - m} = \frac{n^m}{n^m} = 1$$

In definitiva, data la formula generale:

$n^m = n * n^{m - 1}$,

possiamo dire che la condizione di uscita dalla ricorsione e':

$(m - 1) = 0$, e cioe' $m = 1$.

In questo caso infatti si ha:

$n^{m - 1} = n^{1 - 1} = n^0 = 1$.

Vediamo ad esempio come si calcola ricorsivamente la potenza con esponente **4** del numero **2**:

$$2^4 = 2 * 2^{4 - 1} = 2 * 2^3$$
$$2^3 = 2 * 2^{3 - 1} = 2 * 2^2$$
$$2^2 = 2 * 2^{2 - 1} = 2 * 2^1$$
$$2^1 = 2 * 2^{1 - 1} = 2 * 2^0$$
$$2^0 = 1$$

A questo punto abbiamo ottenuto tutti i fattori necessari, per cui, procedendo a ritroso si ottiene:

$1 * 2 * 2 * 2 * 2 = 16$.

La ricorsione nei linguaggi di programmazione.

I linguaggi di programmazione che supportano la ricorsione, permettono la definizione di sottoprogrammi ricorsivi; un sottoprogramma si definisce ricorsivo quando contiene una chiamata (diretta o indiretta) a se stesso. Si ha la ricorsione diretta quando un sottoprogramma chiama direttamente se stesso; si ha invece la ricorsione indiretta quando un sottoprogramma **A** chiama un sottoprogramma **B** che a sua volta chiama di nuovo il sottoprogramma **A**. In generale, dati **n** sottoprogrammi chiamati **A1**, **A2**, **A3**, ..., **An**, si ha la ricorsione indiretta quando **A1** chiama **A2** che chiama **A3**, e cosi' via, sino ad **An** che chiama di nuovo **A1**; si puo' osservare che nel caso di ricorsione diretta, un sottoprogramma ricorsivo svolge allo stesso tempo il ruolo di caller e di called.
Tra i casi piu' importanti di linguaggi di programmazione che supportano la ricorsione troviamo ad esempio il **C** e il **Pascal**; la ricorsione invece non viene supportata dalle vecchie versioni del **BASIC** e dal **FORTRAN77**.
In Figura 1 vediamo l'implementazione non ricorsiva di un sottoprogramma che calcola il fattoriale di un numero **n**.

Figura 1 - Calcolo non ricorsivo di n!

Versione C

```
long fattoriale(int n)
{
   long  i = 1;

   while (n > 1)
      i *= n--;

   return i;
}
```

Versione Pascal

```
function fattoriale(n: Integer):Longint;

var   i: Longint;

begin
   i:= 1;

   while (n > 1) do begin
      i:= i * n;
      n:= n - 1;
   end;

   fattoriale:= i;
end;
```

In Figura 2 vediamo invece l'implementazione ricorsiva di un sottoprogramma che calcola il fattoriale di un numero **n**.

Figura 2 - Calcolo ricorsivo di n!

Versione C

```
long fattoriale(int n)
{
   if (n == 0)
      return 1;
   else
      return (n * fattoriale(n - 1));
}
```

Versione Pascal

```
function fattoriale(n: Integer):Longint;
begin
   if (n = 0) then
      fattoriale:= 1
   else
      fattoriale:= n * fattoriale(n - 1);
end;
```

Nei sistemi operativi a **16** bit, il tipo **long** del **C** e il tipo **Longint** del **Pascal** rappresentano entrambi un intero con segno a **32** bit; nel calcolo del fattoriale di un numero **n**, bisogna tener presente che anche per valori di **n** relativamente piccoli, si possono ottenere come risultato numeri piuttosto grandi. Ad esempio, gia' **9!** supera abbondantemente il valore massimo (**65535**) rappresentabile con **16** bit; analogamente, **13!** supera il valore massimo (**4294967295**) rappresentabile con **32** bit.

In Figura 3 vediamo l'implementazione non ricorsiva di un sottoprogramma che calcola la potenza con esponente **m** di un numero **n**.

Figura 3 - Calcolo non ricorsivo di n^m

Versione C	Versione Pascal

<pre>long potenza(int n, int m)
{
   long i = 1;
 	
   while (m-- > 0)
      i *= n;
 	
   return i;
}</pre>
	<pre>function potenza(n, m: Integer):Longint;
	
	var i: Longint;
	
	begin
	   i:= 1;
	
	   while (m > 0) do begin
	      i:= i * n;
	      m:= m - 1;
	   end;
	
	   potenza:= i;
	end;</pre>

In Figura 4 vediamo invece l'implementazione ricorsiva di un sottoprogramma che calcola la potenza con esponente **m** di un numero **n**.

Figura 4 - Calcolo ricorsivo di n^m

Versione C	Versione Pascal

<pre>long potenza(int n, int m)
{
   if (m == 0)
      return 1;
   else
      return (n * potenza(n, m - 1));
}</pre>
	<pre>function potenza(n, m: Integer):Longint;
	begin
	   if (m = 0) then
	      potenza:= 1
	   else
	      potenza:= n * potenza(n, m - 1);
	end;</pre>

In generale, si nota subito che la versione ricorsiva di un generico algoritmo, e' piu' compatta e piu' comprensibile della versione non ricorsiva; si vede anche che l'implementazione pratica di un algoritmo ricorsivo rispecchia fedelmente la formulazione matematica dello stesso algoritmo.

La ricorsione in Assembly.

Analizzando la Figura 2 e la Figura 4, si puo' avere l'impressione che la ricorsione sia una sorta di fenomeno "paranormale"; questa impressione e' legata al fatto che un sottoprogramma ricorsivo sembra in grado di fornirci "magicamente" il risultato esatto dopo aver chiamato ripetutamente se stesso e senza aver effettuato apparentemente nessun calcolo. In realta', le proprieta' paranormali e le magie, ovviamente non c'entrano niente; vediamo allora con l'ausilio dell'Assembly, come puo' essere svelato il "mistero" della ricorsione.
Supponiamo di voler implementare una procedura ricorsiva dotata di argomenti, variabili locali e valore di ritorno; il problema fondamentale che dobbiamo affrontare, e' legato al metodo che bisogna seguire per garantire che ogni volta che la procedura chiama se stessa, vengano create nuove copie degli argomenti e delle variabili locali che non vadano a sovrascrivere le copie precedenti. Per chiarire questo aspetto, vediamo un esempio pratico in Assembly, relativo al calcolo ricorsivo del fattoriale di un numero **n**; scriviamo una procedura **fatt** che riceve in **BX** un numero intero senza segno a **16** bit, e restituisce in **EAX** il risultato a **32** bit. In base a quanto detto in precedenza, per ottenere un risultato valido dobbiamo inserire in **BX** valori compresi tra **0** e **12**;

inoltre, i **16** bit piu' significativi di **EBX** devono valere zero. La prima versione di questa procedura, ispirata all'esempio di Figura 2, e' la seguente:

```
0064h                   fatt proc near
0064h  66h B8h 00000001h   mov      eax, 1             ; eax = 1
006Ah  85h DBh             test     bx, bx             ; bx = 0 ?
006Ch  74h 07h             jz       short exit_fatt    ; return eax = 1
006Eh  4Bh                 dec      bx                 ; bx = bx - 1
006Fh  E8h FFF2h           call     near ptr fatt      ; ricorsione
0072h  66h F7h E3h         mul      ebx                ; edx:eax = eax * ebx
0075h                   exit_fatt:

0075h  C3h                 ret

0076h                   fatt endp
```

Sulla sinistra notiamo che per ogni istruzione e' presente il relativo offset e il codice macchina prodotto dall'assembler; da queste informazioni si ricava che la procedura **fatt** viene definita all'offset **0064h** di un segmento di codice. Analizziamo in dettaglio quello che succede nel caso in cui **fatt** venga chiamata con **(E)BX=3**; in questo caso, **fatt** dovrebbe restituirci **EAX=6**. Per rendere piu' chiara la spiegazione, viene mostrato passo per passo lo stato dello stack del programma; le condizioni iniziali sono rappresentate da **SP=0200h**, mentre l'indirizzo di ritorno e' **0031h**.
La CPU incontra nel programma principale una istruzione di chiamata **NEAR** a **fatt**, ed esegue quindi i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=019Eh**.
***** Salva l'indirizzo di ritorno **0031h** in **SS:SP=SS:019Eh**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h
Offset	SP=019Eh

Appena entrati in **fatt**, abbiamo quindi: **SP=019Eh** e **(E)BX=3**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** e' diverso da zero, la CPU salta all'istruzione **DEC** ed esegue i seguenti passi:
***** Sottrae uno a **BX** e ottiene **BX=2**.
***** Sottrae due byte a **SP** e ottiene **SP=019Ch**.
***** Salva l'indirizzo di ritorno **0072h** in **SS:SP=SS:019Ch**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h	0072h
Offset	019Eh	SP=019Ch

Appena entrati ricorsivamente in **fatt**, abbiamo: **SP=019Ch** e **(E)BX=2**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** e' diverso da zero, la CPU salta all'istruzione **DEC** ed esegue i seguenti passi:
***** Sottrae uno a **BX** e ottiene **BX=1**.
***** Sottrae due byte a **SP** e ottiene **SP=019Ah**.
***** Salva l'indirizzo di ritorno **0072h** in **SS:SP=SS:019Ah**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h	0072h	0072h
Offset	019Eh	019Ch	SP=019Ah

Appena entrati ricorsivamente in **fatt**, abbiamo: **SP=019Ah** e **(E)BX=1**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** e' diverso da zero, la CPU salta all'istruzione **DEC** ed esegue i seguenti passi:
***** Sottrae uno a **BX** e ottiene **BX=0**.
***** Sottrae due byte a **SP** e ottiene **SP=0198h**.
***** Salva l'indirizzo di ritorno **0072h** in **SS:SP=SS:0198h**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h	0072h	0072h	0072h
Offset	019Eh	019Ch	019Ah	SP=0198h

Appena entrati ricorsivamente in **fatt**, abbiamo: **SP=0198h** e **(E)BX=0**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** vale zero, la CPU salta all'etichetta **exit_fatt**, e incontrando l'istruzione **RET** di tipo **NEAR**, esegue i seguenti passi:
***** Estrae da **SP=0198h** l'indirizzo di ritorno **0072h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=019Ah**.
***** Salta a **CS:IP=CS:0072h** con valore di ritorno **EAX=1**.

Contenuto	0031h	0072h	0072h
Offset	019Eh	019Ch	SP=019Ah

All'indirizzo **CS:0072h** troviamo l'istruzione **MUL**, e in quel momento abbiamo: **SP=019Ah**, **EAX=1** e **(E)BX=0**; l'istruzione **MUL** moltiplica **EAX=1** per **EBX=0** e ottiene **EDX:EAX=0**. Subito dopo la CPU incontra l'istruzione **RET** di tipo **NEAR** ed esegue i seguenti passi:
***** Estrae da **SP=019Ah** l'indirizzo di ritorno **0072h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=019Ch**.
***** Salta a **CS:IP=CS:0072h** con valore di ritorno **EAX=0**.

Contenuto	0031h	0072h
Offset	019Eh	SP=019Ch

All'indirizzo **CS:0072h** troviamo l'istruzione **MUL**, e in quel momento abbiamo: **SP=019Ch**, **EAX=0** e **(E)BX=0**; l'istruzione **MUL** moltiplica **EAX=0** per **EBX=0** e ottiene **EDX:EAX=0**. Subito dopo la CPU incontra l'istruzione **RET** di tipo **NEAR** ed esegue i seguenti passi:
***** Estrae da **SP=019Ch** l'indirizzo di ritorno **0072h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=019Eh**.
***** Salta a **CS:IP=CS:0072h** con valore di ritorno **EAX=0**.

Contenuto	0031h
Offset	SP=019Eh

All'indirizzo **CS:0072h** troviamo l'istruzione **MUL**, e in quel momento abbiamo: **SP=019Eh**, **EAX=0** e **(E)BX=0**; l'istruzione **MUL** moltiplica **EAX=0** per **EBX=0** e ottiene **EDX:EAX=0**. Subito dopo la CPU incontra l'istruzione **RET** di tipo **NEAR** ed esegue i seguenti passi:
***** Estrae da **SP=019Eh** l'indirizzo di ritorno **0031h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=0200h**.
***** Salta a **CS:IP=CS:0031h** con valore di ritorno **EAX=0**.

Il programma principale riottiene il controllo e trova in **EAX** il valore zero che chiaramente non rappresenta il risultato corretto; ripetendo l'esperimento con altri valori passati a **fatt** nel registro **(E)BX**, alla fine si ottiene sempre **EAX=0**. Come si spiega questa situazione?
Analizzando il meccanismo ricorsivo appena descritto, si intuisce subito che il problema risiede nel fatto che ad ogni chiamata ricorsiva di **fatt**, il registro **BX** viene decrementato di uno senza preservare pero' il suo contenuto originario; dopo l'ultima chiamata ricorsiva, **BX** ormai vale zero, per cui la moltiplicazione produce un risultato nullo. Tutte le successive moltiplicazioni tra **EAX** e **EBX** producono di conseguenza un risultato nullo in quanto il fattore **EBX** vale zero; dopo l'ultima moltiplicazione, il risultato nullo che si trova in **EAX** e viene restituito al programma principale.
La soluzione a questo problema, ci viene suggerita dal meccanismo appena descritto, che permette alla CPU di gestire correttamente nello stack gli indirizzi di ritorno relativi alle varie chiamate ricorsive di **fatt**; grazie a questo meccanismo, i vari indirizzi di ritorno vanno ad occupare posizioni differenti nello stack, evitando cosi' di sovrapporsi l'uno all'altro. Possiamo utilizzare allora questa stessa tecnica per gestire nello stack la sequenza di valori che assume **BX** nel corso della ricorsione; in questo modo, possiamo evitare che le modifiche apportate al contenuto di **BX** si trasmettano da una chiamata all'altra di **fatt**.
In base alle considerazioni appena esposte, possiamo dire che la procedura **fatt**, subito dopo l'istruzione **JZ**, deve svolgere i seguenti passi:

***** Salvare il contenuto corrente di **BX**.
***** Decrementare il contenuto corrente di **BX**.
***** Chiamare ricorsivamente **fatt**.
***** Ripristinare il vecchio contenuto di **BX**.

In seguito a queste modifiche, la procedura **fatt** assume il seguente aspetto:

```
0064h                   fatt proc near

0064h  66h B8h 00000001h   mov      eax, 1            ; eax = 1
006Ah  85h DBh             test     bx, bx            ; bx = 0 ?
006Ch  74h 09h             jz       short exit_fatt   ; return eax = 1
006Eh  53h                 push     bx                ; preserva bx
006Fh  4Bh                 dec      bx                ; bx = bx - 1
0070h  E8h FFF1h           call     near ptr fatt     ; ricorsione
0073h  5Bh                 pop      bx                ; ripristina bx
0074h  66h F7h E3h         mul      ebx               ; edx:eax = eax * ebx
0077h                   exit_fatt:

0077h  C3h                 ret

0078h                   fatt endp
```

Proviamo a ripetere l'esperimento, e vediamo cosa succede chiamando **fatt** con **(E)BX=3**; come al solito, supponiamo che la situazione iniziale dello stack sia **SP=0200h**, e che l'indirizzo di ritorno al caller sia **0031h**.
La CPU incontra nel programma principale una istruzione di chiamata **NEAR** a **fatt**, ed esegue quindi i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=019Eh**.
***** Salva l'indirizzo di ritorno **0031h** in **SS:SP=SS:019Eh**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h
Offset	SP=019Eh

Appena entrati in **fatt**, abbiamo quindi: **SP=019Eh** e **(E)BX=3**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** e' diverso da zero, la CPU salta all'istruzione **PUSH** ed esegue i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=019Ch**.
***** Salva il valore **3** di **BX** in **SS:SP=SS:019Ch**.
***** Sottrae uno a **BX** e ottiene **BX=2**.
***** Sottrae due byte a **SP** e ottiene **SP=019Ah**.
***** Salva l'indirizzo di ritorno **0073h** in **SS:SP=SS:019Ah**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h	0003h	0073h
Offset	019Eh	019Ch	SP=019Ah

Appena entrati ricorsivamente in **fatt**, abbiamo: **SP=019Ah** e **(E)BX=2**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** e' diverso da zero, la CPU salta all'istruzione **PUSH** ed esegue i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0198h**.
***** Salva il valore **2** di **BX** in **SS:SP=SS:0198h**.
***** Sottrae uno a **BX** e ottiene **BX=1**.
***** Sottrae due byte a **SP** e ottiene **SP=0196h**.
***** Salva l'indirizzo di ritorno **0073h** in **SS:SP=SS:0196h**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h	0003h	0073h	0002h	0073h
Offset	019Eh	019Ch	019Ah	0198h	SP=0196h

Appena entrati ricorsivamente in **fatt**, abbiamo: **SP=0196h** e **(E)BX=1**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** e' diverso da zero, la CPU salta all'istruzione **PUSH** ed esegue i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0194h**.
***** Salva il valore **1** di **BX** in **SS:SP=SS:0194h**.
***** Sottrae uno a **BX** e ottiene **BX=0**.
***** Sottrae due byte a **SP** e ottiene **SP=0192h**.
***** Salva l'indirizzo di ritorno **0073h** in **SS:SP=SS:0192h**.
***** Carica in **IP** l'offset **0064h** di **fatt** e salta a **CS:IP=CS:0064h**.

Contenuto	0031h	0003h	0073h	0002h	0073h	0001h	0073h
Offset	019Eh	019Ch	019Ah	0198h	0196h	0194h	SP=0192h

Appena entrati ricorsivamente in **fatt**, abbiamo: **SP=0192h** e **(E)BX=0**, mentre in **EAX** viene caricato il valore **1**; siccome **BX** vale zero, la CPU salta all'etichetta **exit_fatt**, e incontrando l'istruzione **RET** di tipo **NEAR**, esegue i seguenti passi:
***** Estrae da **SP=0192h** l'indirizzo di ritorno **0073h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=0194h**.
***** Salta a **CS:IP=CS:0073h** con valore di ritorno **EAX=1**.

Contenuto	0031h	0003h	0073h	0002h	0073h	0001h
Offset	019Eh	019Ch	019Ah	0198h	0196h	SP=0194h

All'indirizzo **CS:0073h** troviamo l'istruzione **POP**, e in quel momento abbiamo: **SP=0194h**, **EAX=1** e **(E)BX=0**; la CPU esegue quindi i seguenti passi:
***** Estrae da **SP=0194h** il valore **0001h** e lo carica in **BX**.
***** Somma due byte a **SP** e ottiene **SP=0196h**.
***** Moltiplica **EAX=1** per **EBX=1** e ottiene **EDX:EAX=1**.
Subito dopo la CPU incontra un'istruzione **RET** di tipo **NEAR** ed esegue i seguenti passi:
***** Estrae da **SP=0196h** l'indirizzo di ritorno **0073h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=0198h**.
***** Salta a **CS:IP=CS:0073h** con valore di ritorno **EAX=1**.

Contenuto	0031h	0003h	0073h	0002h
Offset	019Eh	019Ch	019Ah	SP=0198h

All'indirizzo **CS:0073h** troviamo l'istruzione **POP**, e in quel momento abbiamo: **SP=0198h**, **EAX=1** e **(E)BX=1**; la CPU esegue quindi i seguenti passi:
***** Estrae da **SP=0198h** il valore **0002h** e lo carica in **BX**.
***** Somma due byte a **SP** e ottiene **SP=019Ah**.
***** Moltiplica **EAX=1** per **EBX=2** e ottiene **EDX:EAX=2**.
Subito dopo la CPU incontra un'istruzione **RET** di tipo **NEAR** ed esegue i seguenti passi:
***** Estrae da **SP=019Ah** l'indirizzo di ritorno **0073h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=019Ch**.
***** Salta a **CS:IP=CS:0073h** con valore di ritorno **EAX=2**.

Contenuto	0031h	0003h
Offset	019Eh	SP=019Ch

All'indirizzo **CS:0073h** troviamo l'istruzione **POP**, e in quel momento abbiamo: **SP=019Ch**, **EAX=2** e **(E)BX=2**; la CPU esegue quindi i seguenti passi:
***** Estrae da **SP=019Ch** il valore **0003h** e lo carica in **BX**.
***** Somma due byte a **SP** e ottiene **SP=019Eh**.
***** Moltiplica **EAX=2** per **EBX=3** e ottiene **EDX:EAX=6**.
Subito dopo la CPU incontra un'istruzione **RET** di tipo **NEAR** ed esegue i seguenti passi:
***** Estrae da **SP=019Eh** l'indirizzo di ritorno **0031h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=0200h**.
***** Salta a **CS:IP=CS:0031h** con valore di ritorno **EAX=6**.

Questa volta il programma principale riottiene il controllo e trova in **EAX** il valore **6** che e' ovviamente il risultato corretto; si puo' verificare che **fatt** calcola correttamente il fattoriale di tutti i numeri compresi tra **0** e **12**. Grazie all'uso dello stack, la procedura **fatt** puo' preservare il contenuto di **BX** prima di decrementarlo e passarlo ricorsivamente a se stessa; ad ogni chiamata ricorsiva, la procedura **fatt** gestisce un valore di **BX** indipendente dal valore relativo alle altre chiamate.

Il metodo appena illustrato, utilizza i registri per il passaggio degli argomenti alla procedura **fatt**; come gia' sappiamo, questo metodo consente di ottenere la massima velocita' nel passaggio degli argomenti ad una procedura. L'aspetto negativo e' rappresentato dal fatto che in questo modo stiamo impegnando diversi registri che potrebbero servirci per altri scopi; la situazione poi tende a complicarsi quando utilizziamo ulteriori registri per definire anche variabili locali all'interno di una procedura.

Per ovviare a tutti questi inconvenienti, possiamo sfruttare i concetti esposti nel precedente capitolo in relazione alle convenzioni adottate dai linguaggi di alto livello per il passaggio degli argomenti alle procedure, per la creazione delle variabili locali e per i valori di ritorno; vediamo allora come possiamo implementare la procedura **fatt** seguendo le convenzioni dei compilatori **C** e **Pascal**.
La procedura **fatt** in versione **C** assume l'aspetto seguente:

```
; long fatt(int n)

fatt proc near

   n        equ   [bp+4]      ; parametro n

   push     bp                ; preserva bp
   mov      bp, sp            ; ss:bp = ss:sp

   mov      eax, 1            ; eax = 1
   cmp      word ptr n, 0     ; n = 0 ?
   je       short exit_fatt   ; return eax = 1
   mov      ax, n             ; ax = n
   dec      ax                ; ax = n - 1
   push     ax                ; argomento (n - 1)
   call     near ptr fatt     ; ricorsione
   add      sp, 2             ; ripulisce lo stack
   movzx    edx, word ptr n   ; edx = n
   mul      edx               ; edx:eax = eax * edx
exit_fatt:

   pop      bp                ; ripristina bp
   ret                        ; near ret

fatt endp
```

Il programma principale, prima di chiamare **fatt** deve inserire nello stack un valore compreso tra **0** e **12**, del quale vogliamo calcolare il fattoriale; subito dopo avviene la chaimata di **fatt**. All'interno di **fatt**, il parametro **n** viene letto dallo stack all'indirizzo **SS:BP+4**; questo perche' **fatt** e' una procedura di tipo **NEAR**. Prima di chiamare ricorsivamente **fatt** dobbiamo inserire nello stack una copia dell'argomento **n** decrementato di **1**; a tale proposito viene utilizzato il registro **AX** per evitare di modificare l'argomento **n** originale presente nello stack. Subito dopo la chiamata ricorsiva di **fatt** dobbiamo ripulire lo stack sommando **2** byte a **SP**; ricordiamo infatti che secondo la convenzione **C**, il caller e' tenuto a ripulire lo stack dagli argomenti passati ad una procedura. Questo stesso lavoro deve essere effettuato dal programma principale una volta che ha riottenuto il controllo.

La procedura **fatt** in versione **Pascal** assume l'aspetto seguente:

```
; function fatt(n: Integer):Longint;

fatt proc near

   n        equ   [bp+4]      ; parametro n

   push     bp                ; preserva bp
   mov      bp, sp            ; ss:bp = ss:sp

   mov      eax, 1            ; eax = 1
   cmp      word ptr n, 0     ; n = 0 ?
   je       short exit_fatt   ; return eax = 1
   mov      ax, n             ; ax = n
   dec      ax                ; ax = n - 1
```

```
   push     ax                 ; argomento (n - 1)
   call     near ptr fatt      ; ricorsione
   movzx    edx, word ptr n    ; edx = n
   mul      edx                ; edx:eax = eax * edx
exit_fatt:

   pop      bp                 ; ripristina bp
   ret      (2)                ; near ret e pulizia stack

fatt endp
```

L'unica differenza rispetto alla versione **C** e' rappresentata dal fatto che secondo la convenzione **Pascal**, e' compito del called ripulire lo stack dagli argomenti passati ad una procedura; a tale proposito, la procedura **fatt** utilizza l'istruzione **RET** con l'operando immediato **2** che somma **2** byte a **SP** prima di restituire il controllo al caller.

Se vogliamo interfacciare queste procedure con i compilatori **C** e **Pascal** a **16** bit, dobbiamo ricordarci che i valori di ritorno a **32** bit devono essere restituiti in **DX:AX** e non in **EAX**; a tale proposito possiamo utilizzare l'istruzione:

```
shld edx, eax, 16.
```

Come e' stato spiegato nel Capitolo 18, questa istruzione fa scorrere di **16** posti verso sinistra i bit di **EDX**; i **16** posti rimasti liberi a destra vengono riempiti con i **16** bit piu' significativi di **EAX**, mentre il contenuto di **EAX** rimane inalterato. Attraverso questa istruzione, il contenuto di **EAX** viene copiato in **DX:AX**; nel rispetto della convenzione **little endian**, il registro **DX** contiene la word piu' significativa di **EAX**. Ulteriori dettagli su questi argomenti vengono illustrati nei capitoli dedicati all'interfacciamento tra l'Assembly e i linguaggi di alto livello.

Analizziamo in dettaglio un ulteriore esempio rappresentato dalla versione **Pascal** di una procedura chiamata **Potenza**, che calcola ricorsivamente la potenza di ordine **m** di un numero **n**; questa procedura assume il seguente aspetto:

```
; function Potenza(n, m: Integer):Longint;

007Ch                      Potenza proc near

                              n        equ    [bp+6]      ; parametro n
                              m        equ    [bp+4]      ; parametro m

007Ch  55h                    push     bp                 ; preserva bp
007Dh  8Bh ECh                mov      bp, sp             ; ss:bp = ss:sp

007Fh  66h B8h 00000001h      mov      eax, 1             ; eax = 1
0085h  83h 7Eh 04h 00h        cmp      word ptr m, 0      ; m = 0 ?
0089h  74h 13h                je       short exit_proc    ; return eax = 1
008Bh  FFh 76h 06h            push     word ptr n         ; push n
008Eh  8Bh 46h 04h            mov      ax, m              ; ax = m
0091h  48h                    dec      ax                 ; ax = m - 1
0092h  50h                    push     ax                 ; push m - 1
0093h  E8h FFE6h              call     near ptr Potenza   ; chiamata ricorsiva
0096h  66h 0Fh B7h 56h 06h    movzx    edx, word ptr n    ; edx = n
009Bh  66h F7h E2h            mul      edx                ; edx:eax = eax * edx
009Eh                      exit_proc:

009Eh  5Dh                    pop      bp                 ; ripristina bp
009Fh  C2h 0004h              ret      4                  ; near ret e pulizia
stack

00A2h                      Potenza endp
```

Vediamo ora quello che succede quando il programma principale chiama **Potenza** con argomenti **base=2** e **esp=3**; se la procedura non contiene errori, dovrebbe restituirci in **EAX** il valore **8** (**2^3**). Supponiamo che le condizioni iniziali dello stack siano rappresentate da **SP=0200h**, e che l'indirizzo di ritorno al caller sia **0033h**.

La CPU incontra nel programma principale la seguente sequenza di istruzioni:

```
; Potenza(base, esp)

push     base
push     esp
call     near ptr Potenza
```

ed esegue quindi i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=019Eh**.
***** Salva il valore **2** (copia di **base**) in **SS:SP=SS:019Eh**.
***** Sottrae due byte a **SP** e ottiene **SP=019Ch**.
***** Salva il valore **3** (copia di **esp**) in **SS:SP=SS:019Ch**.
***** Sottrae due byte a **SP** e ottiene **SP=019Ah**.
***** Salva l'indirizzo di ritorno **0033h** in **SS:SP=SS:019Ah**.
***** Carica in **IP** l'offset **007Ch** di **Potenza** e salta a **CS:IP=CS:007Ch**.

Appena entrati in **Potenza**, incontriamo l'istruzione **PUSH bp** che preserva nello stack il contenuto originario di **BP** (che indichiamo con **oldBP1**); la CPU esegue quindi i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0198h**.
***** Salva il valore **oldBP1** in **SS:SP=SS:0198h**.

Contenuto	0002h	0003h	0033h	oldBP1
Offset	019Eh	019Ch	019Ah	SP=0198h

Subito dopo, il contenuto (**0198h**) di **SP** viene copiato in **BP**, per cui possiamo dire che il parametro **n=2** si trova in **SS:BP+6**, il parametro **m=3** si trova in **SS:BP+4**, l'indirizzo di ritorno **0033h** si trova in **SS:BP+2**, mentre **oldBP1** si trova in **SS:BP**. A questo punto abbiamo quindi: **SP=0198h**, **n=2** e **m=3**, mentre in **EAX** viene caricato il valore **1**; siccome **m** e' diverso da zero, la CPU salta all'istruzione **PUSH n** ed esegue i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0196h**.
***** Salva il valore **2** (copia di **n**) in **SS:SP=SS:0196h**.
***** Decrementa la copia **AX** di **m** e ottiene **AX=2**.
***** Sottrae due byte a **SP** e ottiene **SP=0194h**.
***** Salva il valore **AX=2** in **SS:SP=SS:0194h**.
***** Sottrae due byte a **SP** e ottiene **SP=0192h**.
***** Salva l'indirizzo di ritorno **0096h** in **SS:SP=SS:0192h**.
***** Carica in **IP** l'offset **007Ch** di **Potenza** e salta a **CS:IP=CS:007Ch**.

Appena entrati ricorsivamente in **Potenza**, incontriamo l'istruzione **PUSH bp** che preserva nello stack il contenuto originario di **BP** (che indichiamo con **oldBP2**); la CPU esegue quindi i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0190h**.
***** Salva il valore **oldBP2** in **SS:SP=SS:0190h**.

Contenuto	0002h	0003h	0033h	oldBP1	0002h	0002h	0096h	oldBP2
Offset	019Eh	019Ch	019Ah	0198h	0196h	0194h	0192h	SP=0190h

Subito dopo, il contenuto (**0190h**) di **SP** viene copiato in **BP**, per cui possiamo dire che il parametro **n=2** si trova in **SS:BP+6**, il parametro **m=2** si trova in **SS:BP+4**, l'indirizzo di ritorno **0096h** si trova in **SS:BP+2**, mentre **oldBP2** si trova in **SS:BP**. A questo punto abbiamo quindi: **SP=0190h**, **n=2** e **m=2**, mentre in **EAX** viene caricato il valore **1**; siccome **m** e' diverso da zero, la CPU salta all'istruzione **PUSH n** ed esegue i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=018Eh**.
***** Salva il valore **2** (copia di **n**) in **SS:SP=SS:018Eh**.
***** Decrementa la copia **AX** di **m** e ottiene **AX=1**.
***** Sottrae due byte a **SP** e ottiene **SP=018Ch**.
***** Salva il valore **AX=1** in **SS:SP=SS:018Ch**.
***** Sottrae due byte a **SP** e ottiene **SP=018Ah**.
***** Salva l'indirizzo di ritorno **0096h** in **SS:SP=SS:018Ah**.
***** Carica in **IP** l'offset **007Ch** di **Potenza** e salta a **CS:IP=CS:007Ch**.

Appena entrati ricorsivamente in **Potenza**, incontriamo l'istruzione **PUSH bp** che preserva nello stack il contenuto originario di **BP** (che indichiamo con **oldBP3**); la CPU esegue quindi i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0188h**.
***** Salva il valore **oldBP3** in **SS:SP=SS:0188h**.

Contenuto	0002h	0003h	0033h	oldBP1	0002h	0002h	0096h	oldBP2	0002h	0001h	0096h	oldBP3
Offset	019Eh	019Ch	019Ah	0198h	0196h	0194h	0192h	0190h	018Eh	018Ch	018Ah	SP=0188h

Subito dopo, il contenuto (**0188h**) di **SP** viene copiato in **BP**, per cui possiamo dire che il parametro **n=2** si trova in **SS:BP+6**, il parametro **m=1** si trova in **SS:BP+4**, l'indirizzo di ritorno **0096h** si trova in **SS:BP+2**, mentre **oldBP3** si trova in **SS:BP**. A questo punto abbiamo quindi: **SP=0188h**, **n=2** e **m=1**, mentre in **EAX** viene caricato il valore **1**; siccome **m** e' diverso da zero, la CPU salta all'istruzione **PUSH n** ed esegue i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0186h**.
***** Salva il valore **2** (copia di **n**) in **SS:SP=SS:0186h**.
***** Decrementa la copia **AX** di **m** e ottiene **AX=0**.
***** Sottrae due byte a **SP** e ottiene **SP=0184h**.
***** Salva il valore **AX=0** in **SS:SP=SS:0184h**.
***** Sottrae due byte a **SP** e ottiene **SP=0182h**.
***** Salva l'indirizzo di ritorno **0096h** in **SS:SP=SS:0182h**.
***** Carica in **IP** l'offset **007Ch** di **Potenza** e salta a **CS:IP=CS:007Ch**.

Appena entrati ricorsivamente in **Potenza**, incontriamo l'istruzione **PUSH bp** che preserva nello stack il contenuto originario di **BP** (che indichiamo con **oldBP4**); la CPU esegue quindi i seguenti passi:
***** Sottrae due byte a **SP** e ottiene **SP=0180h**.
***** Salva il valore **oldBP4** in **SS:SP=SS:0180h**.

Contenuto	0002h	0003h	0033h	oldBP1	0002h	0002h	0096h	oldBP2	0002h	0001h	0096h	oldBP3	0002h	0000h	0096h	oldBP4
Offset	019Eh	019Ch	019Ah	0198h	0196h	0194h	0192h	0190h	018Eh	018Ch	018Ah	0188h	0186h	0184h	0182h	SP=0180h

Subito dopo, il contenuto (**0180h**) di **SP** viene copiato in **BP**, per cui possiamo dire che il parametro **n=2** si trova in **SS:BP+6**, il parametro **m=0** si trova in **SS:BP+4**, l'indirizzo di ritorno **0096h** si trova in **SS:BP+2**, mentre **oldBP4** si trova in **SS:BP**. A questo punto abbiamo quindi: **SP=0180h**, **n=2** e **m=0**, mentre in **EAX** viene caricato il valore **1**; siccome **m** vale zero, la CPU salta all'etichetta **exit_proc**, dove incontra l'istruzione **POP bp** seguita dall'istruzione **RET** di tipo **NEAR** con operando immediato **4**. La CPU esegue quindi i seguenti passi:
***** Estrae da **SP=0180h** il valore **oldBP4** e lo carica in **BP**.
***** Somma due byte a **SP** e ottiene **SP=0182h**.
***** Estrae da **SP=0182h** l'indirizzo di ritorno **0096h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=0184h**.
***** Somma quattro byte a **SP** e ottiene **SP=0188h**.
***** Salta a **CS:IP=CS:0096h** con valore di ritorno **EAX=1**.

Contenuto	0002h	0003h	0033h	oldBP1	0002h	0002h	0096h	oldBP2	0002h	0001h	0096h	oldBP3
Offset	019Eh	019Ch	019Ah	0198h	0196h	0194h	0192h	0190h	018Eh	018Ch	018Ah	SP=0188h

All'indirizzo **CS:0096h** troviamo l'istruzione **MOVZX**, e in quel momento abbiamo: **SP=0188h**, **EAX=1**, **n=2** e **m=1**; la CPU esegue quindi i seguenti passi:
***** Carica **n=2** in **EDX**.
***** Moltiplica **EAX=1** per **EDX=2** e ottiene **EDX:EAX=2**.
Subito dopo la CPU incontra l'istruzione **POP bp** seguita dall'istruzione **RET** di tipo **NEAR** con operando immediato **4** ed esegue i seguenti passi:
***** Estrae da **SP=0188h** il valore **oldBP3** e lo carica in **BP**.
***** Somma due byte a **SP** e ottiene **SP=018Ah**.
***** Estrae da **SP=018Ah** l'indirizzo di ritorno **0096h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=018Ch**.
***** Somma quattro byte a **SP** e ottiene **SP=0190h**.
***** Salta a **CS:IP=CS:0096h** con valore di ritorno **EAX=2**.

Contenuto	0002h	0003h	0033h	oldBP1	0002h	0002h	0096h	oldBP2
Offset	019Eh	019Ch	019Ah	0198h	0196h	0194h	0192h	SP=0190h

All'indirizzo **CS:0096h** troviamo l'istruzione **MOVZX**, e in quel momento abbiamo: **SP=0190h**, **EAX=2**, **n=2** e **m=2**; la CPU esegue quindi i seguenti passi:
***** Carica **n=2** in **EDX**.
***** Moltiplica **EAX=2** per **EDX=2** e ottiene **EDX:EAX=4**.
Subito dopo la CPU incontra l'istruzione **POP bp** seguita dall'istruzione **RET** di tipo **NEAR** con operando immediato **4** ed esegue i seguenti passi:
***** Estrae da **SP=0190h** il valore **oldBP2** e lo carica in **BP**.
***** Somma due byte a **SP** e ottiene **SP=0192h**.
***** Estrae da **SP=0192h** l'indirizzo di ritorno **0096h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=0194h**.
***** Somma quattro byte a **SP** e ottiene **SP=0198h**.
***** Salta a **CS:IP=CS:0096h** con valore di ritorno **EAX=4**.

Contenuto	0002h	0003h	0033h	oldBP1
Offset	019Eh	019Ch	019Ah	SP=0198h

All'indirizzo **CS:0096h** troviamo l'istruzione **MOVZX**, e in quel momento abbiamo: **SP=0198h**, **EAX=4**, **n=2** e **m=3**; la CPU esegue quindi i seguenti passi:
***** Carica **n=2** in **EDX**.
***** Moltiplica **EAX=4** per **EDX=2** e ottiene **EDX:EAX=8**.
Subito dopo la CPU incontra l'istruzione **POP bp** seguita dall'istruzione **RET** di tipo **NEAR** con operando immediato **4** ed esegue i seguenti passi:
***** Estrae da **SP=0198h** il valore **oldBP1** e lo carica in **BP**.
***** Somma due byte a **SP** e ottiene **SP=019Ah**.
***** Estrae da **SP=019Ah** l'indirizzo di ritorno **0033h** e lo carica in **IP**.
***** Somma due byte a **SP** e ottiene **SP=019Ch**.
***** Somma quattro byte a **SP** e ottiene **SP=0200h**.
***** Salta a **CS:IP=CS:0033h** con valore di ritorno **EAX=8**.

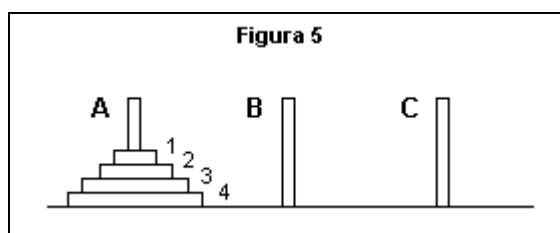
Il programma principale riottiene il controllo e trova in **EAX** il valore **8** che e' proprio il risultato di 2^3; osserviamo che per ottenere un risultato valido, n^m non deve superare il massimo valore gestibile da **EAX**, e cioe' **4294967295** ($2^{32}-1$). A differenza di quanto accade con il **C**, il **Pascal** non supporta gli interi senza segno a **32** bit, ma solo gli interi con segno a **32** bit (**Longint**); il **Pascal** dispone invece del tipo **comp** che e' un intero con segno a **64** bit, che puo' essere restituito da una procedura attraverso la cima dello stack (**ST(0)**) della **FPU**.
Dall'esempio appena svolto, possiamo dedurre alcune considerazioni importanti. Prima di tutto osserviamo che le strutture dati di tipo **LIFO** (Last In First Out) come lo stack, si prestano in modo perfetto per le procedure ricorsive; l'uso dello stack quindi consente di implementare in modo semplice ed efficiente gli algoritmi ricorsivi. Affinche' il meccanismo appena descritto funzioni in modo perfetto, e' fondamentale che il contenuto originario del registro **BP** venga rigorosamente preservato; non ci vuole molto a capire che in caso contrario, il programma andrebbe immediatamente in crash.

Vantaggi e svantaggi delle procedure ricorsive.

Attraverso i vari esempi illustrati in precedenza, possiamo analizzare i vantaggi e gli svantaggi delle procedure ricorsive.
Il vantaggio fondamentale della ricorsione e' rappresentato dall'estrema compattezza del codice scritto con questa tecnica; tale compattezza diventa particolarmente evidente nel caso di algoritmi di complessita' medio alta (come viene mostrato piu' avanti).
Abbiamo anche visto che un algoritmo ricorsivo, esteriormente ha una struttura molto piu' semplice e comprensibile del corrispondente algoritmo non ricorsivo; e' chiaro pero' che per riuscire a risolvere ricorsivamente problemi molto complessi, bisognerebbe essere dei geni della matematica. Questo non significa che gli algoritmi non ricorsivi siano piu' semplici da individuare rispetto a quelli ricorsivi; anzi, nella maggior parte dei casi e' vero l'opposto.
In generale, possiamo dire che gli svantaggi presentati dalla ricorsione superano abbondantemente i (pochissimi) vantaggi; di fatto, la possibilita' di scrivere codice molto compatto e' l'unico vantaggio degno di nota della ricorsione.
Analizzando i vari esempi di questo capitolo, si nota subito che le procedure in versione non ricorsiva permettono di spingere al massimo la sequenzialita' del codice; come gia' sappiamo, piu' il codice e' sequenziale, piu' sara' eseguito velocemente dalla CPU. Le corrispondenti procedure in versione ricorsiva invece, chiamano ripetutamente se stesse con sensibili ripercussioni sulle prestazioni dovute al tempo necessario per il trasferimento del controllo; la conseguenza di tutto cio' e' data dal fatto che in generale, la versione non ricorsiva di una procedura e' molto piu' veloce della corrispondente versione ricorsiva.
Il principale difetto delle procedure ricorsive e' rappresentato sicuramente dal notevolissimo consumo di spazio nello stack; si tratta di un aspetto estremamente pericoloso in quanto in molti casi puo' portare alla saturazione di tutto il segmento di stack del nostro programma. In un caso del genere, ulteriori chiamate ricorsive comportano la scrittura di dati in aree della memoria che non appartengono allo stack; nei casi estremi vengono sovrascritte aree della memoria riservate al sistema operativo, con conseguente blocco del computer!
Per evidenziare questo aspetto, possiamo osservare quello che accade nel caso della procedura **Potenza**; la versione non ricorsiva di questa procedura richiede uno spazio minimo nello stack pari a **8** byte. Questi **8** byte sono necessari per contenere l'indirizzo di ritorno (**2** byte), il parametro **n** (**2** byte), il parametro **m** (**2** byte) e il contenuto originario di **BP** (**2** byte); se si decide inoltre di preservare i vari registri generali utilizzati dalla procedura, si puo' arrivare a circa **16** byte. Per la versione ricorsiva di **Potenza**, la situazione cambia radicalmente; questa versione infatti ha bisogno di almeno **8** byte per ogni chiamata ricorsiva, piu' **8** byte per la prima chiamata effettuata dal caller.
Nel caso ad esempio di **m=3**, possiamo notare che sono necessari almeno:
`8 * (3 + 1) = 8 * 4 = 32` byte.

Nel caso di **m=20** si arriva ad una richiesta minima di spazio nello stack pari a:
`8 * (20 + 1) = 8 * 21 = 168` byte.
In generale, la procedura ricorsiva **Potenza**, chiamata con l'esponente **m**, richiede un minimo di:
`8 * (m + 1)` byte di spazio nello stack.
Si tenga anche presente che **Potenza** e' una procedura estremamente semplice; nel caso di procedure molto complesse, la situazione puo' facilmente degenerare nella completa saturazione dello stack.
Proprio questa caratteristica molto pericolosa, spinse i progettisti della **IBM** a non supportare la ricorsione nel linguaggio **FORTRAN77**; si tratta di un linguaggio di programmazione nato intorno al **1954**, e destinato alla risoluzione dei problemi matematici con il computer. **FORTRAN** infatti e' la contrazione di **IBM Mathematical Formula Translation System**; del resto, le disponibilita' di memoria dei computers dell'epoca non permettevano certo l'implementazione di procedure ricorsive.

Le Torri di Hanoi.

Per evidenziare ulteriormente le caratteristiche positive e negative della ricorsione, analizziamo in dettaglio un celebre algoritmo che permette di risolvere ricorsivamente un famoso gioco chiamato: **Le Torri di Hanoi**; si tratta di un antico gioco dell'estremo oriente, praticato dai monaci buddisti.
La Figura 5 mostra una rappresentazione schematica di questo gioco; l'obiettivo e' quello di trasferire tutti i dischi da palo **A** al palo **C** servendosi del palo **B**.

Le regole del gioco sono le seguenti:

***** Si puo' spostare un solo disco alla volta.
***** Non si puo' sistemare un disco sopra un'altro di diametro inferiore.
***** Non si puo' sistemare nessun disco fuori dai tre pali.

La risoluzione di questo problema con un algoritmo non ricorsivo si presenta estremamente difficile; viceversa, il problema puo' essere risolto con un algoritmo ricorsivo caratterizzato da una disarmante semplicita'.
Cominciamo con due soli dischi indicati con **1** e **2**, e osserviamo che in questo caso, il gioco si risolve in modo molto intuitivo con le seguenti tre mosse:

```
1) Spostare il disco 1 da A a B
2) Spostare il disco 2 da A a C
3) Spostare il disco 1 da B a C
```

In pratica, spostiamo il disco **1** dal palo di partenza al palo intermedio; poi spostiamo l'unico disco rimasto (il numero **2**), dal palo di partenza al palo di arrivo, e infine spostiamo il disco **1** dal palo intermedio al palo di arrivo. Queste tre mosse sono molto importanti perche' rappresentano la condizione alla quale dobbiamo cercare di pervenire nel caso di **3** o piu' dischi; nel caso di **2** dischi, la sequenza delle tre mosse puo' essere generalizzata in questo modo:

```
1) Spostare 2-1 dischi (1 disco) dal palo di partenza al palo di arrivo
servendosi del palo intermedio
2) Spostare direttamente il disco rimasto, dal palo di partenza al palo di
arrivo
3) Spostare 2-1 dischi (1 disco) dal palo intermedio al palo di arrivo
servendosi del palo di partenza
```

Vediamo come si puo' applicare in pratica questo algoritmo; nel caso di **3** dischi possiamo scrivere le tre mosse:

```
1) Spostare 3-1 dischi (2 dischi) dal palo A al palo B, servendosi del palo C
2) Spostare direttamente il disco rimasto, dal palo A al palo C
3) Spostare 3-1 dischi (2 dischi) dal palo B al palo C, servendosi del palo A
```

Siccome non siamo in grado di effettuare le mosse **1** e **3**, dobbiamo cercare di scomporle in due o piu' mosse, sino a pervenire al caso (che sappiamo risolvere) di due soli dischi; osserviamo subito che la mossa:

```
1) Spostare 3-1 dischi (2 dischi) dal palo A al palo B, servendosi del palo C,
```

puo' essere scomposta nelle seguenti mosse:

```
1a) Spostare 2-1 dischi (1 disco) dal palo A al palo C, servendosi del palo B
2a) Spostare direttamente il disco rimasto, dal palo A al palo B
3a) Spostare 2-1 dischi (1 disco) dal palo C al palo B, servendosi del palo A
```

Analogamente, la mossa:

```
3) Spostare 3-1 dischi (2 dischi) dal palo B al palo C, servendosi del palo A,
```

puo' essere scomposta nelle seguenti mosse:

```
1b) Spostare 2-1 dischi (1 disco) dal palo B al palo A, servendosi del palo C
2b) Spostare direttamente il disco rimasto, dal palo B al palo C
3b) Spostare 2-1 dischi (1 disco) dal palo A al palo C, servendosi del palo B
```

A questo punto possiamo dire che nel caso di **3** dischi, il gioco puo essere risolto con la seguente sequenza di **7** mosse:

```
1) Spostare un disco dal palo A al palo C
2) Spostare un disco dal palo A al palo B
3) Spostare un disco dal palo C al palo B
4) Spostare un disco dal palo A al palo C
5) Spostare un disco dal palo B al palo A
6) Spostare un disco dal palo B al palo C
7) Spostare un disco dal palo A al palo C
```

Nel caso di **4** dischi possiamo scrivere le tre mosse:

```
1) Spostare 4-1 dischi (3 dischi) dal palo A al palo B, servendosi del palo C
2) Spostare direttamente il disco rimasto, dal palo A al palo C
3) Spostare 4-1 dischi (3 dischi) dal palo B al palo C, servendosi del palo A
```

Siccome non siamo in grado di effettuare le mosse **1** e **3**, dobbiamo cercare di scomporle in due o piu' mosse, sino a pervenire al caso (che sappiamo risolvere) di due soli dischi; osserviamo subito che la mossa:

```
1) Spostare 4-1 dischi (3 dischi) dal palo A al palo B, servendosi del palo C,
```

puo' essere scomposta nelle seguenti mosse:

```
1a) Spostare 3-1 dischi (2 dischi) dal palo A al palo C, servendosi del palo B
2a) Spostare direttamente il disco rimasto, dal palo A al palo B
3a) Spostare 3-1 dischi (2 dischi) dal palo C al palo B, servendosi del palo A
```

Analogamente, la mossa:

```
3) Spostare 4-1 dischi (3 dischi) dal palo B al palo C, servendosi del palo A,
```

puo' essere scomposta nelle seguenti mosse:

```
1b) Spostare 3-1 dischi (2 dischi) dal palo B al palo A, servendosi del palo C
2b) Spostare direttamente il disco rimasto, dal palo B al palo C
3b) Spostare 3-1 dischi (2 dischi) dal palo A al palo C, servendosi del palo B
```

Le mosse **1a**, **3a**, **1b** e **3b** possono essere risolte con il metodo gia' spiegato per il caso di **3** dischi; alla fine, per il caso di **4** dischi, si ottiene la soluzione rappresentata dalla seguente sequenza di **15** mosse:

```
 1) Spostare un disco dal palo A al palo B
 2) Spostare un disco dal palo A al palo C
 3) Spostare un disco dal palo B al palo C
 4) Spostare un disco dal palo A al palo B
 5) Spostare un disco dal palo C al palo A
 6) Spostare un disco dal palo C al palo B
 7) Spostare un disco dal palo A al palo B
 8) Spostare un disco dal palo A al palo C
 9) Spostare un disco dal palo B al palo C
10) Spostare un disco dal palo B al palo A
11) Spostare un disco dal palo C al palo A
12) Spostare un disco dal palo B al palo C
13) Spostare un disco dal palo A al palo B
14) Spostare un disco dal palo A al palo C
15) Spostare un disco dal palo B al palo C
```

A questo punto appare evidente che nel caso generale di **N** dischi possiamo scrivere le tre mosse:

```
1) Spostare N-1 dischi dal palo A al palo B, servendosi del palo C
2) Spostare direttamente il disco rimasto, dal palo A al palo C
3) Spostare N-1 dischi dal palo B al palo C, servendosi del palo A
```

Questo algoritmo afferma che per spostare **N** dischi dal palo **A** al palo **C**, dobbiamo prima spostare **N-1** dischi dal palo **A** al palo **B** attraverso il palo **C**; subito dopo dobbiamo spostare direttamente il disco rimasto, dal palo **A** al palo **C**, e infine dobbiamo spostare **N-1** dischi dal palo **B** al palo **C** attraverso il palo **A**. Appare evidente il fatto che questo e' un classico caso di algoritmo ricorsivo;

indicando quindi con **N** il numero di dischi, con **A** il palo di partenza, con **C** il palo di arrivo e con **B** il palo intermedio, possiamo schematizzare questo algoritmo con il seguente pseudocodice:

```
Hanoi(N, A, C, B)
inizio
se (N = 0)
   esci;
altrimenti
   Hanoi(N-1, A, B, C);
   Sposta un disco da A a C
   Hanoi(N-1, B, C, A);
fine
```

La condizione di uscita dalla ricorsione e' rappresentata da **N=0**; infatti, quando **N** diventa zero, vuol dire che non ci sono piu' dischi da spostare. Puo' sembrare incredibile, ma questo (apparentemente) insignificante algoritmo funziona in modo perfetto; in Figura 6 vediamo una implementazione reale scritta in Linguaggio **C**.

Figura 6 - HANOI.C

```
/*************************************************/
/* file hanoi.c                                  */
/* Soluzione ricorsiva delle Torri di Hanoi      */
/*************************************************/

#include < stdio.h >

/* prototipi di funzione */

void hanoi(int, char, char, char);

/* entry point */

int main(void)
{
   int      num = 1;

   while (num != 0)
   {
      printf("Inserire il numero dei dischi: ");
      scanf("%d", &num);
      hanoi(num, 'A', 'C', 'B');
   }

   return 0;
}

/* hanoi: soluzione ricorsiva delle Torri di Hanoi */

void hanoi(int n, char dalPalo, char alPalo, char viaPalo)
{
   if (n == 0)
      return;

   hanoi(n - 1, dalPalo, viaPalo, alPalo);
   printf("Spostare un disco da %c a %c\n", dalPalo, alPalo);
   hanoi(n - 1, viaPalo, alPalo, dalPalo);
}
```

Come si puo' notare, grazie alla sintassi "evoluta" dei linguaggi di alto livello, la struttura della funzione **hanoi** rispecchia fedelmente la formulazione matematica dell'algoritmo ricorsivo; questa funzione e' formata da appena **5** linee di codice **C**, ed evidenzia ulteriormente l'estrema compattezza degli algoritmi ricorsivi, anche nel caso di problemi complessi.

La Figura 7 mostra la versione Assembly del programma che risolve ricorsivamente il gioco delle **Torri di Hanoi**:

Figura 7 - HANOI.ASM

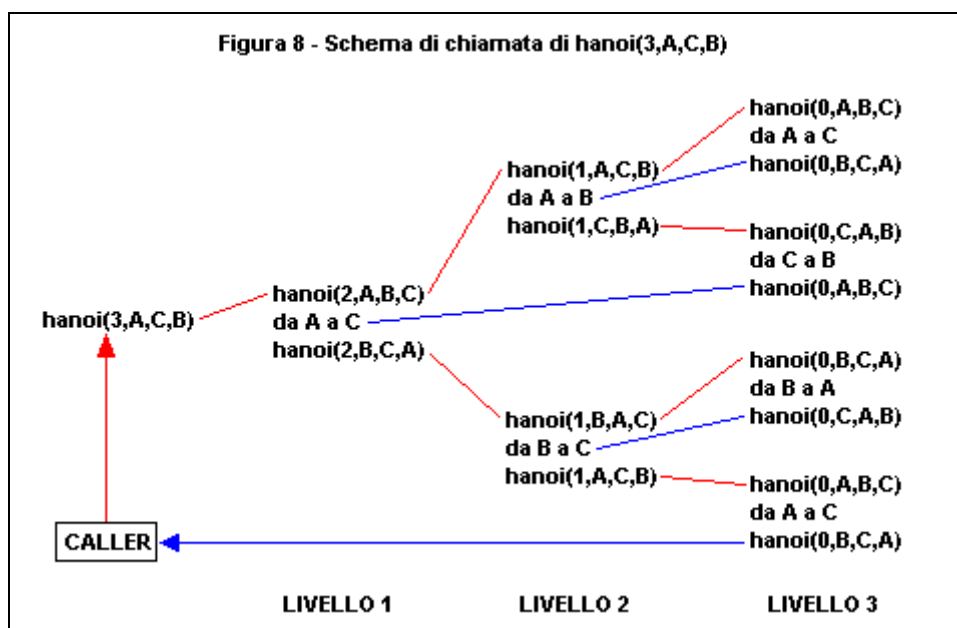
```
;--------------------------------------------------;
; file hanoi.asm                                   ;
; soluzione ricorsiva delle Torri di Hanoi         ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit

INCLUDE     EXELIB.INC              ; libreria di I/O

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                  ; 1024 byte per lo stack

;################ segmento dati ##################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

num         dw    1
messaggio   db    'Sposta un disco da X a Y', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM    ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   call     set80x50                ; modo testo 80 x 50 a 16 colori
   call     hideCursor              ; nasconde il cursore

while_loop:
   cmp      num, 0                  ; num == 0 ?
   je       short exit_while        ; fine ciclo while
```

```
   call     readUdec16            ; legge un intero a 16 bit
   mov      num, ax               ; e lo salva in num
   call     clearScreen           ; cancella lo schermo
   mov      bl, 'A'               ; bl = 'A' (palo di partenza)
   mov      cl, 'C'               ; cl = 'C' (palo di arrivo)
   mov      dl, 'B'               ; dl = 'B' (palo intermedio)

   push     dx                    ; push 'B'
   push     cx                    ; push 'C'
   push     bx                    ; push 'A'
   push     ax                    ; push num
   call     near ptr hanoi        ; hanoi(num, 'A', 'C', 'B');
   add      sp, 8                 ; pulizia dello stack

   jmp      while_loop            ; ricomincia
exit_while:

   call     showCursor            ; visualizza il cursore
   call     set80x25              ; modo testo 80 x 25 a 16 colori

;-------- fine blocco principale istruzioni -------

   mov      al, 00h               ; codice di uscita
   mov      ah, 4ch               ; proc. return to DOS
   int      21h                   ; chiama i servizi DOS

;------------- inizio blocco procedure ------------

;-----------------------------------------------------------------;
; void hanoi(int n, char dalPalo, char alPalo, char viaPalo)     ;
;-----------------------------------------------------------------;
; convenzioni del linguaggio C per il passaggio degli argomenti ;
; e per la pulizia dello stack                                   ;
;-----------------------------------------------------------------;

hanoi proc near

   n        equ   [bp+4]          ; parametro n
   dalPalo  equ   [bp+6]          ; parametro dalPalo
   alPalo   equ   [bp+8]          ; parametro alPalo
   viaPalo  equ   [bp+10]         ; parametro viaPalo

   push     bp                    ; preserva bp
   mov      bp, sp                ; ss:bp = ss:sp

   cmp      word ptr n, 0         ; if (n == 0)
   je       exit_hanoi            ;    esci
   dec      word ptr n            ; n = n - 1

   push     word ptr alPalo
   push     word ptr viaPalo
   push     word ptr dalPalo
   push     word ptr n
   call     near ptr hanoi        ; hanoi(n-1, dalPalo, viaPalo, alPalo)
   add      sp, 8                 ; pulizia stack

   mov      bx, offset messaggio  ; bx punta a messaggio
   mov      al, dalPalo
   mov      [bx+19], al           ; messaggio[19] = dalPalo
   mov      al, alPalo
   mov      [bx+23], al           ; messaggio[23] = alPalo
   mov      dx, bx                ; dx punta a messaggio
   mov      ah, 09h               ; servizio DOS Display String
```

```
   int      21h                    ; visualizza la stringa

   push     word ptr dalPalo
   push     word ptr alPalo
   push     word ptr viaPalo
   push     word ptr n
   call     near ptr hanoi         ; hanoi(n-1, viaPalo, alPalo, dalPalo)
   add      sp, 8                  ; pulizia stack

exit_hanoi:

   pop      bp                     ; ripristina bp
   ret                             ; near return

hanoi endp

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)

STACKSEGM   ENDS

;##################################################

   END      start
```

completa su questo algoritmo, bisogna verificare anche altri aspetti, come la velocita' e la richiesta di spazio nello stack.
Una caratteristica piuttosto evidente della procedura **hanoi**, e' data dal fatto che al crescere del numero di dischi da spostare, cresce in modo impressionante il numero delle mosse necessarie; parallelamente si registra un sensibile calo della velocita' di esecuzione, legato evidentemente alla notevole crescita del numero di chiamate ricorsive. Per analizzare in dettaglio questi aspetti, possiamo cercare prima di tutto di individuare la relazione che esiste tra il numero **N** di dischi da spostare, e il numero **M** di mosse necessarie; in termini matematici, si tratta di determinare la funzione:
$M = f(N)$.
Eseguendo il programma **HANOI**, possiamo ricavare questa funzione in modo empirico; osserviamo subito che nel caso di **N=1**, viene stampata una sola mossa, e quindi possiamo scrivere:
$1 = 2^1 - 1$.
Nel caso di **N=2**, vengono stampate **3** mosse, e quindi possiamo scrivere:
$3 = 2^2 - 1$.
Nel caso di **N=3**, vengono stampate **7** mosse, e quindi possiamo scrivere:
$7 = 2^3 - 1$.
Nel caso di **N=4**, vengono stampate **15** mosse, e quindi possiamo scrivere:
$15 = 2^4 - 1$.
A questo punto possiamo dire con certezza che nel caso generale di **N** dischi, verra' stampato un numero di mosse:
$M = 2^N - 1$.
Questa relazione ci dice che al crescere del numero **N** di dischi, il numero **M** di mosse cresce in modo direttamente proporzionale alla potenza con esponente **N** di **2**; tutto cio' giustifica il fatto che quando si va' oltre i **10** dischi, si comincia a notare un sensibile rallentamento del programma,

anche con **CPU** piuttosto potenti. Nel caso ad esempio di **N=16**, sono necessarie ben:
`2^16 - 1 = 65535` mosse!

Queste considerazioni sono piuttosto preoccupanti se pensiamo che per ciascuna chiamata della procedura **hanoi**, sono necessari almeno **12** byte di stack (**2** per il parametro **n**, **2** per **dalPalo**, **2** per **alPalo**, **2** per **viaPalo**, **2** per l'indirizzo di ritorno e **2** per **BP**); questo significa che se il numero di chiamate ricorsive fosse pari al numero di mosse, con **N=16** avremmo bisogno di ben:
`65535 * 12 = 786420` byte di stack!
Fortunatamente pero' le cose non stanno esattamente cosi'; per rendercene conto, possiamo provare a seguire il percorso delle varie chiamate ricorsive di **hanoi**. La Figura 8 mostra lo schema di chiamata di **hanoi** nel caso di **N=3**; le linee rosse indicano i percorsi di andata, mentre le linee blu indicano i percorsi di ritorno.

* Il programma principale (**CALLER**) chiama **hanoi(3,A,C,B)** con inserimento nello stack di **12** byte.
* **hanoi(3,A,C,B)** chiama **hanoi(2,A,B,C)** con inserimento nello stack di altri **12** byte per un totale di **24** byte.
* **hanoi(2,A,B,C)** chiama **hanoi(1,A,C,B)** con inserimento nello stack di altri **12** byte per un totale di **36** byte.
* **hanoi(1,A,C,B)** chiama **hanoi(0,A,B,C)** con inserimento nello stack di altri **12** byte per un totale di **48** byte.
* **hanoi(0,A,B,C)** termina (**N=0**) liberando **12** byte nello stack, per un totale di **36** byte.
* Viene stampata la stringa **"da A a C"**.
* Viene chiamata **hanoi(0,B,C,A)** con inserimento nello stack di altri **12** byte per un totale di **48** byte.
* **hanoi(0,B,C,A)** termina (**N=0**) liberando **12** byte nello stack, per un totale di **36** byte, e restituisce il controllo a **hanoi(1,A,C,B)**.
* **hanoi(1,A,C,B)** termina liberando **12** byte nello stack, per un totale di **24** byte.
* Viene stampata la stringa **"da A a B"**.
* Viene chiamata **hanoi(1,C,B,A)** con inserimento nello stack di altri **12** byte per un totale di **36** byte.
* **hanoi(1,C,B,A)** chiama **hanoi(0,C,A,B)** con inserimento nello stack di altri **12** byte per un totale di **48** byte.

*** hanoi(0,C,A,B)** termina (**N=0**) liberando **12** byte nello stack, per un totale di **36** byte.
***** Viene stampata la stringa **"da C a B"**.
***** Viene chiamata **hanoi(0,A,B,C)** con inserimento nello stack di altri **12** byte per un totale di **48** byte.
*** hanoi(0,A,B,C)** termina (**N=0**) liberando **12** byte nello stack, per un totale di **36** byte, e restituisce il controllo a **hanoi(1,C,B,A)**.
*** hanoi(1,C,B,A)** termina liberando **12** byte nello stack, per un totale di **24** byte, e restituisce il controllo a **hanoi(2,A,B,C)**.
*** hanoi(2,A,B,C)** termina liberando **12** byte nello stack, per un totale di **12** byte.
***** Viene stampata la stringa **"da A a C"**.
***** Viene chiamata **hanoi(2,B,C,A)** con inserimento nello stack di altri **12** byte per un totale di **24** byte.
*** hanoi(2,B,C,A)** chiama **hanoi(1,B,A,C)** con inserimento nello stack di altri **12** byte per un totale di **36** byte.

A questo punto e' perfettamente inutile continuare in quanto si capisce subito che con **N=3**, non verranno mai superati i **48** byte di spazio necessario nello stack; questi **48** byte sono relativi quindi alla prima chiamata di **hanoi** effettuata dal **CALLER**, e ai tre livelli di ricorsione indicati in Figura 8 con **LIVELLO 1**, **LIVELLO 2** e **LIVELLO 3**. In pratica, la massima occupazione di spazio nello stack verra' raggiunta quando **N** diventa zero.
Ripetendo lo schema di Figura 8 per diversi valori di **N**, si puo' constatare che:

con **N=1** sono necessari **24** byte di stack, e cioe':
`12 * (1 + 1)` (con **1** livello di ricorsione),
con **N=2** sono necessari **36** byte di stack, e cioe':
`12 * (2 + 1)` (con **2** livelli di ricorsione),
con **N=3** sono necessari **48** byte di stack, e cioe':
`12 * (3 + 1)` (con **3** livelli di ricorsione),
con **N=4** sono necessari **60** byte di stack, e cioe':
`12 * (4 + 1)` (con **4** livelli di ricorsione),
e cosi' via.
Generalizzando quindi, possiamo dire che se vogliamo spostare **N** dischi, abbiamo bisogno come minimo di:
`12 * (N + 1)` byte di spazio nello stack.
Nel caso ad esempio di **N=16**, ci servono "solamente" **204** byte e non **786420**.

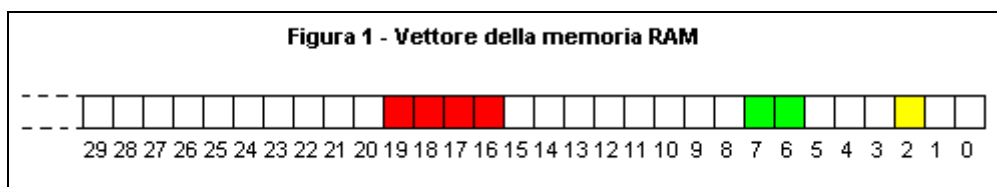
L'esempio delle **Torri di Hanoi**, ci fa capire ulteriormente quanto sia importante l'analisi dettagliata del comportamento di un determinato algoritmo ricorsivo; in questo modo e' possibile evitare spiacevoli sorprese in fase di esecuzione di quei programmi che fanno uso della ricorsione. Si tenga presente che lo **stack overflow** (superamento della capacita' massima del segmento di stack), e' uno dei bugs piu' pericolosi e piu' difficili da scovare; proprio per questo motivo, i compilatori dei linguaggi di alto livello offrono al programmatore la possibilita' di inserire automaticamente nei programmi, il codice necessario per la verifica dello stack overflow.

Gli algoritmi ricorsivi rappresentano un settore della matematica che con l'avvento del computer ha accresciuto enormemente la sua importanza; infatti, moltissimi problemi matematici possono essere formultati in modo ricorsivo. La ricorsione viene largamente applicata anche nei programmi che sfidano gli esseri umani nel gioco degli scacchi; per ulteriori approfondimenti si puo' fare riferimento ai numerosi libri che trattano questo argomento.

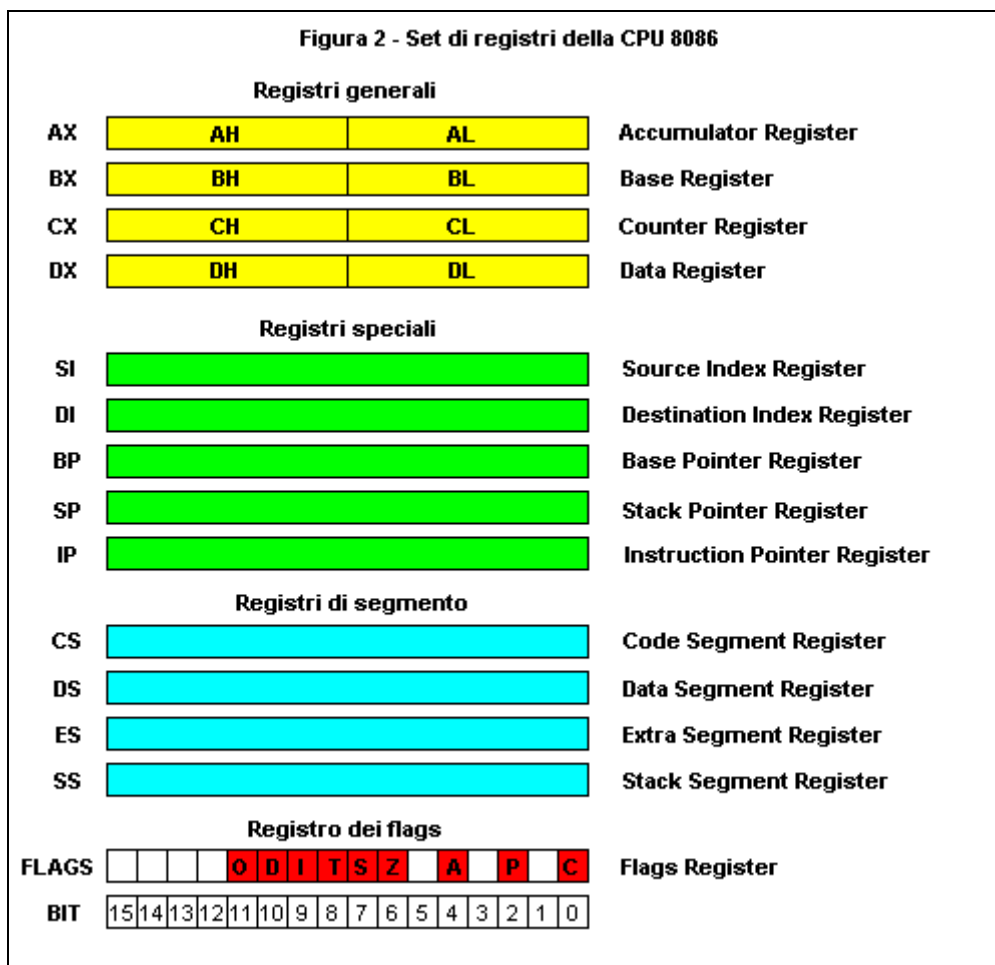
Capitolo 27 - I modelli di memoria

Nei precedenti capitoli e' stato detto che in generale, un programma scritto in un qualunque linguaggio di programmazione, puo' essere suddiviso in tre blocchi fondamentali destinati a contenere il **codice**, i **dati** e lo **stack**; il blocco stack contiene i dati temporanei (dinamici) del programma, il blocco dati contiene i dati statici del programma, mentre il blocco codice contiene le istruzioni che elaborano i dati. Nel caso piu' semplice un programma ha bisogno di un solo blocco per il codice, un solo blocco per i dati e un solo blocco per lo stack; puo' anche capitare pero' che un programma abbia la necessita' di disporre di piu' blocchi di codice e/o piu' blocchi di dati. In relazione a queste necessita' un programma puo' assumere tutta una serie di possibili configurazioni distinte, corrispondenti a diverse esigenze di memoria; nella terminologia dei linguaggi di alto livello, queste configurazioni vengono anche chiamate **modelli di memoria**. In questo capitolo vengono illustrate in dettaglio le principali configurazioni che puo' assumere un programma Assembly; tutte le considerazioni che seguono sono riferite come al solito alla modalita' di indirizzamento reale **8086**.

La modalita' reale 8086.

Per una migliore comprensione degli argomenti esposti in questo capitolo, puo' essere utile un richiamo dei principali concetti legati alla modalita' di indirizzamento reale **8086**.
Ogni volta che si accende il computer, tutte le CPU della famiglia **80x86** vengono inizializzate in modalita' di emulazione dell'**8086**; in questa modalita' viene garantita la compatibilita' verso il basso, nel senso che ad esempio, anche una CPU di classe **80586**, **80686**, etc, puo' eseguire codice **8086**. Per ottenere questa compatibilita', le CPU **80x86** simulano l'architettura a **16** bit, tipica dell'**8086**; riepiloghiamo quindi le caratteristiche principali della CPU **8086**.
Tutte le CPU della famiglia **80x86** trattano la memoria **RAM** come se fosse un vettore di byte (Figura 1); gli indici dei vari byte iniziano da zero, per cui il primo byte ha indice **0**, il secondo byte ha indice **1**, il terzo byte ha indice **2** e cosi' via.

L'indice di un determinato byte della **RAM**, coincide con il suo indirizzo fisico (**physical address**); possiamo dire quindi che il byte di indice **0** si trova all'indirizzo fisico **0** della **RAM**, il byte di indice **1** si trova all'indirizzo fisico **1** della **RAM** e cosi' via. Un blocco di memoria **RAM** composto da uno o piu' byte consecutivi e contigui forma una **locazione di memoria**; la dimensione in byte di una locazione di memoria coincide ovviamente con il numero di byte che la compongono.
L'indirizzo fisico di una locazione di memoria coincide con l'indirizzo fisico del suo primo byte (byte meno significativo); in Figura 1 vediamo ad esempio una locazione di memoria da **1** byte (in giallo) che si trova all'indirizzo fisico **2**, una locazione di memoria da **2** byte (in verde) che si trova all'indirizzo fisico **6**, e una locazione di memoria da **4** byte (in rosso) che si trova all'indirizzo fisico **16**. La dimensione in byte di una locazione di memoria determina il massimo valore binario che la locazione stessa puo' contenere; una locazione da **1** byte puo' contenere tutti i numeri binari compresi tra **0** e **255**, una locazione da **2** byte puo' contenere tutti i numeri binari compresi tra **0** e **65535**, una locazione da **4** byte puo' contenere tutti i numeri binari compresi tra **0** e **4294967295**, etc. Da queste considerazioni appare evidente il fatto che ogni locazione di memoria viene univocamente determinata da tre parametri fondamentali; questi tre parametri sono: l'**indirizzo** della locazione, la **dimensione** in byte della locazione e il **contenuto** della locazione.

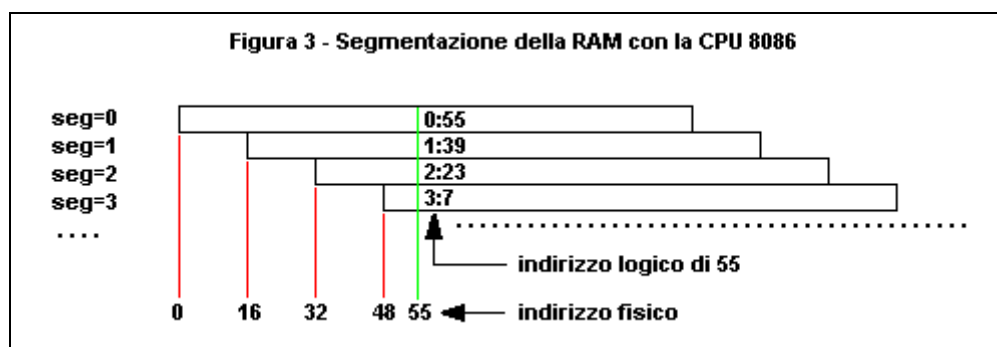
Per poter accedere in lettura e/o in scrittura ad una determinata locazione della **RAM**, la CPU ne deve specificare l'indirizzo fisico; per svolgere questo lavoro, viene utilizzata una connessione elettrica tra la CPU e la **RAM**, che prende il nome di **Address Bus** (bus degli indirizzi). L'**Address Bus** e' formato da un certo numero di linee elettriche che vengono indicate con: **A0**, **A1**, **A2**, etc. In ciascuna di queste linee transita un bit del valore binario che rappresenta l'indirizzo di memoria al quale la CPU vuole accedere; ne consegue che il numero di linee dell'**Address Bus** determina la massima memoria **RAM** fisica che puo' essere indirizzata da una CPU. Nel caso delle CPU **8086**, viene utilizzato un **Address Bus** a **20** linee (da **A0** ad **A19**), capace di gestire quindi indirizzi fisici formati da un massimo di **20** bit; con **20** bit possiamo formare tutti i numeri compresi tra **0** e **1048575**, per un totale di **1048576** valori differenti. In definitiva, le CPU **8086** sono in grado di "vedere" al massimo **1048576** byte di **RAM** fisica, pari a **1** Mb; tutte le CPU di classe **80386** o superiore con architettura a **32** bit, sono dotate di **Address Bus** a **32** linee (da **A0** ad **A31**), attraverso il quale possono indirizzare sino a $2^{32} = 4294967296$ byte di **RAM** fisica (**4** Gb). Quando lavorano in modalita' reale, queste CPU devono tener conto dell'**Address Bus** a **20** linee dell'**8086**; a tale proposito, viene disabilitata la linea **A20** in modo da ottenere un **Address Bus** formato da sole **20** linee, compatibile quindi con quello dell'**8086**.

Un indirizzo fisico inviato attraverso l'**Address Bus**, connette fisicamente la CPU alla corrispondente locazione di memoria; lo scambio di informazioni tra la CPU e la locazione di memoria avviene attraverso una connessione elettrica chiamata **Data Bus** (bus dei dati). Il **Data Bus** e' formato da un certo numero di linee elettriche che vengono indicate con: **D0**, **D1**, **D2**, etc; su ciascuna di queste linee transita un bit dell'informazione che la CPU deve scambiare con la locazione di memoria alla quale e' connessa. Le informazioni vengono gestite dalla CPU a gruppi di bit che sono sempre multipli di **8**; in linea teorica quindi una CPU puo' accedere via hardware ad una informazione formata da **8** bit, **16** bit, **32** bit, **64** bit, etc. In pratica pero', il numero di bit di informazione che la CPU puo' gestire via hardware, ha un limite superiore che rappresenta un parametro importantissimo; questo parametro infatti definisce l'**architettura** della CPU, e cioe' il massimo numero di bit di informazione che la CPU puo' gestire direttamente (via hardware). Possiamo dire quindi che una CPU con architettura a **8** bit e' in grado di gestire via hardware informazioni formate al massimo da **8** bit; in questo caso, tutte le informazioni formate da **16** bit, **32** bit, **64** bit, etc, devono essere gestite via software scomponendole in gruppi di **8** bit. Analogamente, una CPU con architettura a **16** bit e' in grado di gestire via hardware informazioni formate al massimo da **16** bit; in questo caso, tutte le informazioni formate da **32** bit, **64** bit, etc, devono essere gestite via software scomponendole in gruppi di **8** o **16** bit.
L'architettura di una CPU viene fisicamente determinata dalla dimensione massima in bit dei **registri**, che come sappiamo sono delle piccole memorie ad altissima velocita' di accesso, situate all'interno della stessa CPU; la Figura 2 mostra il set di registri in dotazione alla CPU **8086** con architettura a **16** bit.

I **registri generali** sono cosi' chiamati perche' possono essere utilizzati dal programmatore per svolgere svariati compiti; i loro nomi derivano dal ruolo specifico che essi svolgono quando vengono utilizzati come registri predefiniti per determinate istruzioni eseguite dalla CPU. Nei precedenti capitoli abbiamo visto infatti che **AX** (accumulatore) rappresenta il registro preferenziale della CPU, **BX** (indirizzo base) viene largamente utilizzato per indirizzare i dati di un programma, **CX** (contatore) viene usato come contatore nei loop, mentre **DX** (dati) viene spesso utilizzato come registro dati nelle istruzioni aritmetiche; ogni registro generale puo' essere suddiviso in due **half registers** (mezzi registri) che rappresentano la parte alta (**High**) e la parte bassa (**Low**) del registro stesso.
I **registri speciali** vengono utilizzati per l'indirizzamento dei dati statici (**SI**, **DI**), dei dati temporanei (**SP**, **BP**) e delle istruzioni (**IP**) di un programma; il registro **IP** (puntatore alle istruzioni) non e' direttamente accessibile al programmatore.
I **registri di segmento** permettono di suddividere un programma in diversi blocchi attraverso i quali e' possibile separare in modo chiaro ed elegante il codice, i dati e lo stack del programma stesso; i quattro registri di segmento di cui dispone l'**8086** permettono di gestire istante per istante sino a quattro segmenti di programma distinti.
Il **registro dei flags** fornisce sia alla CPU che al programmatore, dettagliate informazioni sul risultato prodotto da una istruzione logico aritmetica appena eseguita; utilizzando queste informazioni e' possibile implementare sofisticate strutture per il controllo del flusso di un programma.

Il concetto fondamentale da ribadire riguarda il fatto che l'architettura di una CPU determina il massimo numero di bit di informazione che la CPU stessa puo' gestire via hardware; con il termine

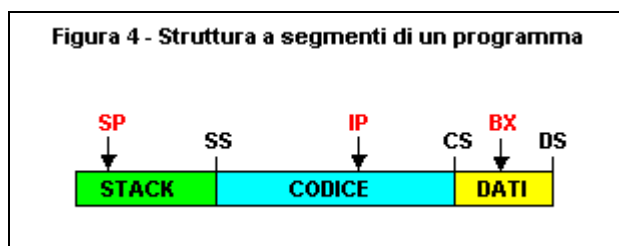
informazione si indica in generale un valore binario che puo' rappresentare un generico numero o un indirizzo di memoria. Nel caso ad esempio dell'**8086**, l'architettura a **16** bit di questa CPU permette di gestire via hardware numeri binari formati al massimo da **16** bit; se con questi numeri binari vogliamo rappresentare indirizzi fisici della **RAM**, ci accorgiamo subito che si viene a creare un evidente problema. Questo problema e' legato al fatto che se vogliamo accedere ad una determinata locazione di memoria, dobbiamo comunicare alla CPU il corrispondente indirizzo fisico; a tale proposito possiamo servirci di appositi registri puntatori come ad esempio **BX**, **SI** e **DI**. Se ad esempio vogliamo accedere attraverso **BX** ad una locazione di memoria che si trova in **RAM** all'indirizzo fisico **3200**, non dobbiamo fare altro che caricare il valore **3200** in **BX**; a questo punto, se richiediamo il trasferimento in **AL** del byte puntato da **BX**, la CPU seleziona attraverso l'**Address Bus** l'indirizzo fisico **3200**, legge **1** byte da questo indirizzo e attraverso il **Data Bus** lo trasferisce in **AL**. Analoghe considerazioni valgono per il caso in cui il byte da leggere si trovi in **RAM** all'indirizzo fisico **60000**; cosa succede pero' se la locazione di memoria a cui vogliamo accedere si trova in **RAM** all'indirizzo fisico **120000**?
Appare evidente che con i registri a **16** bit dell'**8086** possiamo accedere solamente agli indirizzi fisici della **RAM** compresi tra **0** e **65535**; complessivamente con questo metodo e' possibile gestire solo i primi **65536** byte della memoria **RAM**. Per poter gestire tutto il Mb di **RAM** visibile dall'**8086** dobbiamo servirci di un altro metodo; i progettisti dell'**8086** hanno risolto il problema attraverso la "segmentazione" della memoria. In pratica, il Mb di **RAM** dell'**8086** viene suddiviso in tanti blocchi da **65536** byte ciascuno, chiamati **segmenti di memoria**; se vogliamo accedere ad una determinata locazione di memoria, dobbiamo specificare non il suo indirizzo fisico, ma il suo indirizzo logico (**logical address**). Un indirizzo logico e' formato da una coppia di valori a **16** bit che viene chiamata coppia **seg:offset**; la componente **seg** indica il segmento di memoria in cui si trova l'indirizzo fisico a cui vogliamo accedere, mentre la componente **offset** indica la posizione esatta (espressa in byte) dell'indirizzo fisico all'interno del segmento di memoria. In sostanza, **seg=0** rappresenta il primo blocco da **65536** byte di **RAM**, **seg=1** rappresenta il secondo blocco da **65536** byte di **RAM**, etc; in questo modo, l'indirizzo fisico **120000** si verrebbe a trovare a **54464** byte di distanza dall'inizio del secondo segmento di memoria, e quindi si avrebbe: **seg:offset = 1:54464**.
Si puo' osservare che con questo metodo e' possibile gestire una quantita' enorme di memoria fisica; infatti, i **16** bit della componente **seg** ci permettono di rappresentare **65536** segmenti di memoria differenti, ciascuno grande **65536** byte per un totale di:

```
65536 * 65536 = 4294967296 byte = 4 Gb.
```

L'**Address Bus** a **20** linee dell'**8086** permette pero' di indirizzare solo **1** Mb di **RAM** fisica; si rende necessario allora un espediente che permetta di far ricadere in questo unico Mb tutte le possibili coppie **seg:offset**. I progettisti della CPU **8086** hanno allora escogitato il sistema illustrato dalla Figura 3.

Anche in questo caso, sia la componente **seg** che la componente **offset** sono numeri a **16** bit che ci permettono di gestire **65536** segmenti di memoria differenti, ciascuno grande **65536** byte; come si vede in Figura 3 pero', questa volta i vari segmenti di memoria non sono tutti distinti tra loro in quanto vengono fatti partire da indirizzi fisici multipli di **16** byte. Osserviamo infatti che **seg=0** parte dall'indirizzo fisico **0**, **seg=1** parte dall'indirizzo fisico **16**, **seg=2** parte dall'indirizzo fisico **32**, **seg=3** parte dall'indirizzo fisico **48**, **seg=4** parte dall'indirizzo fisico **64**, etc; un blocco di memoria da **16** byte viene definito **paragrafo** di memoria. In pratica, e' come se la memoria venisse suddivisa in **65536** paragrafi; con questo metodo quindi possiamo indirizzare complessivamente:
`65536 * 16 = 1048576 byte = 1 Mb.`
Questo sistema di segmentazione della memoria permette alla CPU di convertire facilmente un indirizzo logico in un indirizzo fisico; osserviamo infatti che essendo **seg** un multiplo di **16**, possiamo ricavare l'indirizzo fisico associato ad una coppia **seg:offset** con la semplice formula:
`indirizzo fisico = (seg * 16) + offset.`
A causa del fatto che i vari segmenti di memoria possono intersecarsi tra loro, uno stesso indirizzo fisico puo' essere associato a diversi indirizzi logici; la corrispondenza tra indirizzi fisici e indirizzi logici non e' quindi biunivoca. In Figura 3 vediamo un esempio pratico relativo a diversi indirizzi logici che rappresentano tutti lo stesso indirizzo fisico **55**; l'indirizzo fisico **55** infatti puo' essere espresso con la coppia **seg=0, offset=55**, oppure con **seg=1, offset=39**, oppure con **seg=2, offset=23**, oppure con **seg=3, offset=7**.

Il sistema di indirizzamento della **RAM** appena descritto rappresenta la **modalita' di indirizzamento reale 8086**; tutte le CPU di classe superiore, quando lavorano in modalita' di emulazione dell'**8086**, devono attenersi a queste regole. Avendo a disposizione una CPU di classe **80386** o superiore, possiamo abilitare l'uso dei registri a **32** bit senza uscire dalla modalita' reale; per compatibilita' con l'**8086** pero', tutti gli indirizzamenti si svolgono ugualmente a **16** bit. In sostanza, una CPU **80386** o superiore che sta lavorando in modalita' reale, quando incontra un'istruzione del tipo:
`mov ax, [ebx],`
utilizza solamente i **16** bit meno significativi di **EBX** per ricavare la componente **offset** dell'indirizzo logico a cui deve accedere; nella sezione **Modalita' Protetta**, viene illustrato un metodo che permette di utilizzare offset a **32** bit in modalita' reale.

Per gestire la modalita' reale in modo semplice ed efficiente, la CPU **8086** mette a disposizione il set di registri illustrato in Figura 2; lo scopo fondamentale di questi registri e' quello di consentire la suddivisione di un programma Assembly in tanti blocchi funzionali chiamati **segmenti di programma** (da non confondere con i segmenti di memoria). Normalmente, ciascun segmento di programma e' destinato a contenere un solo tipo di informazione, cioe' solo codice, oppure solo dati, oppure solo stack; in questo modo si puo' distinguere tra segmenti di codice, segmenti di dati e segmenti di stack. Si tenga presente comunque che in ambiente **DOS**, ciascun segmento di programma e' contemporaneamente leggibile, scrivibile ed eseguibile; questo significa che all'interno di uno stesso segmento di programma possiamo inserire eventualmente un miscuglio di codice, dati e stack.
Tutte le informazioni contenute in un segmento di programma vengono indirizzate attraverso le solite coppie **seg:offset**; la componente **seg** viene assegnata dal **SO**, e rappresenta il paragrafo di memoria da cui inizia effettivamente il segmento di programma. Ricordiamo infatti che un segmento di programma puo' avere l'attributo di allineamento di tipo **BYTE**, **WORD**, **DWORD**, **PARA**, etc; indipendentemente dall'attributo di allineamento, tutti gli **offset** delle informazioni contenute in un segmento di programma, vengono calcolati rispetto al paragrafo di memoria piu' vicino all'indirizzo iniziale del segmento di programma stesso. Osserviamo che la componente **offset** e' un valore a **16** bit che puo' assumere quindi **65536** valori differenti; possiamo dire allora che con gli offset a **16** bit (attributo **USE16**), la dimensione massima di un segmento di programma

non puo' superare **65536** byte.
Attraverso i **4** registri di segmento **CS**, **DS**, **ES** e **SS** della CPU **8086**, possiamo gestire istante per istante sino a **4** segmenti di programma contemporaneamente; ciascuno di questi registri viene inizializzato con la componente **seg** del segmento di programma che vogliamo referenziare. Lo schema predefinito prevede che **CS** referenzi un segmento di codice, **DS** referenzi un segmento di dati e **SS** referenzi un segmento di stack; eventualmente **ES** puo' essere utilizzato per referenziare un segmento extra di dati (con le CPU **80386** e superiori e' possibile referenziare due ulteriori segmenti di dati con **FS** e **GS**). All'interno dei vari segmenti di programma, il ruolo di componente **offset** puo' essere svolto dai registri puntatori dell'**8086**; gli unici registri dell'**8086** che possono essere usati come puntatori sono **BX**, **SI**, **DI**, **SP**, **BP** e **IP**. Il comportamento predefinito della CPU prevede che **IP** venga associato al segmento di programma correntemente referenziato da **CS**; analogamente, **SP** e **BP** vengono associati ad **SS**, mentre **BX**, **SI** e **DI** vengono associati a **DS**. Tutto questo significa che ogni volta che la CPU incontra un indirizzamento effettuato ad esempio con **BX**, ricava da **DS:BX** l'indirizzo logico a cui accedere; in alcuni casi il programmatore puo' alterare questa situazione attraverso il **segment override**.
La Figura 4 mostra la classica struttura segmentata di un programma Assembly formato da un segmento di dati, un segmento di codice e un segmento di stack; la memoria viene rappresentata come al solito con gli indirizzi crescenti da destra verso sinistra.

Come si puo' notare, in questo programma il segmento **DATI** e' stato assegnato a **DS**, il segmento **CODICE** a **CS** e il segmento **STACK** a **SS**; in questo modo la CPU utilizza automaticamente **IP** per muoversi all'interno di **CODICE** (**CS:IP**) e **SP** per muoversi all'interno di **STACK** (**SS:SP**). Inoltre, tutti gli indirizzamenti effettuati con **BX**, **SI** e **DI**, in assenza di segment override vengono associati automaticamente a **DATI**; infatti **DATI** e' referenziato da **DS**, e quindi la CPU ricava gli indirizzi logici dalle coppie **DS:BX**, **DS:SI** e **DS:DI**.

La struttura illustrata in Figura 4, rappresenta il caso piu' frequente per un programma Assembly; questo caso prevede l'utilizzo di un solo segmento di codice, un solo segmento di dati e un solo segmento di stack. Puo' capitare pero' di avere la necessita' di scrivere programmi aventi una struttura piu' semplice o piu' complessa di quella mostrata in Figura 4; in questo caso l'Assembly lascia la massima liberta' d'azione al programmatore. E' possibile infatti creare programmi Assembly aventi le strutture piu' strane e fantasiose; l'importante e' che vengano rispettati i concetti esposti in precedenza. Nel seguito del capitolo vengono illustrati i casi principali che si possono presentare; in particolare vengono analizzati alcuni importantissimi aspetti legati alla presenza di due o piu' segmenti di dati. In tutte le considerazioni che seguono si suppone che il programma sia interamente contenuto in un solo file; nel prossimo capitolo vedremo come distribuire un programma Assembly su piu' files.

Il modello di memoria TINY.

Supponiamo di avere un programma per il quale la somma complessiva delle dimensioni del codice, dei dati e dello stack sia non superiore a **65536** byte; in questo caso esiste la possibilita' di inserire codice, dati e stack in un unico segmento di programma. Cio' e' possibile in quanto, come e' stato detto in precedenza, il **DOS** inizializza il suo ambiente operativo in modo che i programmi siano dotati di segmenti accessibili contemporaneamente in lettura, in scrittura e in esecuzione; questo significa appunto che un segmento di programma **DOS** puo' contenere contemporaneamente codice, dati e stack. Naturalmente sarebbe meglio evitare il piu' possibile questo miscuglio di informazioni; in ogni caso, con un minimo di buon senso e di razionalita' e' possibile scrivere ugualmente programmi "monosegmento" semplici e comprensibili. Nella terminologia dei linguaggi di alto livello, un programma formato da un unico segmento contenente codice, dati e stack, viene definito **programma con modello di memoria TINY**; il termine **TINY** in inglese significa minuscolo.
Il caso piu' impegnativo si presenta quando abbiamo la necessita' di generare un eseguibile in formato **EXE** (EXEcutable); in questo caso infatti il programmatore deve effettuare una serie di inizializzazioni al fine di garantire il corretto funzionamento del programma. La Figura 5 mostra un esempio pratico che chiarisce questo aspetto.

Figura 5 - Modello di memoria TINY (EXE)

```
;---------------------------------------------------;
; File tiny.asm                                     ;
; programma Assembly con modello di memoria TINY    ;
; (formato EXE)                                     ;
;---------------------------------------------------;

;########### direttive per l'assembler ############

.386                               ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                 ; 1024 byte per lo stack

;################ segmento unico ##################

TINYSEGM    SEGMENT  PARA PUBLIC USE16 'SEG_UNICO'

   assume   cs: TINYSEGM, ds: TINYSEGM
   assume   es: TINYSEGM, ss: TINYSEGM

;---------- inizio definizione variabili ----------

stringa1    db    'Chiamata NEAR di printStr1', 10, 13, '$'
stringa2    db    'Chiamata FAR di printStr2', 10, 13, '$'

;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

; void printStr1(char *strOffs)

printStr1 proc near

   strOffs  equ   [bp+4]           ; parametro strOffs

   push     bp                     ; preserva bp
```

```
   mov      bp, sp                  ; ss:bp = ss:sp

   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr1 endp

; void far printStr2(char far *strAddr)
; strAddr = strSeg + strOffs

printStr2 proc far

   strOffs  equu   [bp+6]           ; parametro strOffs
   strSeg   equ    [bp+8]           ; parametro strSeg

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp
   push     ds                      ; preserva ds

   mov      ax, strSeg              ; carica strSeg
   mov      ds, ax                  ; in ds
   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      ds                      ; ripristina ds
   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr2 endp

;-------------- fine blocco procedure -------------

start:                              ; entry point

   cli                              ; disabilita le INT. hardware
   mov      ax, TINYSEGM            ; trasferisce TINYSEGM
   mov      ds, ax                  ; in DS
   mov      es, ax                  ; in ES
   mov      ss, ax                  ; e in SS

   mov      sp, offset stack_label  ; fa puntare SP a stack_label
   sti                              ; riabilita le INT. hardware

;------- inizio blocco principale istruzioni ------

   push     offset stringa1         ; parametro strOffs
   call     near ptr printStr1      ; chiama printStr1 (NEAR)
   add      sp, 2                   ; pulizia stack

   push     seg stringa2            ; parametro strSeg
   push     offset stringa2         ; parametro strOffs
   call     far ptr printStr2       ; chiama printStr2 (FAR)
   add      sp, 4                   ; pulizia stack

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; servizio Return to DOS
```

```
   int      21h                     ; chiama i servizi DOS

;---------- inizio definizione variabili ----------

;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

;-------------- fine blocco procedure -------------

;--------------- inizio blocco stack --------------

   align    2                          ; allineamento alla word

vstack      db    STACK_SIZE dup (?)   ; 1024 byte per lo stack
stack_label:                           ; etichetta di fine stack

;---------------- fine blocco stack ---------------

TINYSEGM    ENDS

;##################################################

   END      start
```

Osserviamo subito che il programma di Figura 5 e' dotato di un unico segmento chiamato **TINYSEGM**; per enfatizzare il fatto che si tratta dell'unico segmento del programma, a **TINYSEGM** e' stato assegnato l'attributo di classe **'SEG_UNICO'**.
In una situazione di questo genere la cosa migliore da fare consiste nell'avere:
CS = DS = ES = SS = TINYSEGM; in questo modo possiamo ottenere notevoli semplificazioni nella gestione del programma. In particolare il fatto che tutti i registri di segmento referenzino lo stesso segmento di programma, ci permette di evitare moltissimi segment override; questo e' vero principalmente per le istruzioni per la manipolazione delle stringhe. Supponiamo ad esempio di voler effettuare un grosso trasferimento dati all'interno di **TINYSEGM** con una istruzione del tipo:
`REP MOVSW`;
incontrando questa istruzione la CPU trasferisce **CX** word da **DS:SI** a **ES:DI**. In questo caso il programmatore deve preoccuparsi solamente dell'inizializzazione di **CX**, **SI** e **DI**; non e' necessaria invece nessuna modifica ai registri di segmento in quanto referenziano tutti l'unico segmento di programma **TINYSEGM** (**CS=DS=ES=SS=TINYSEGM**).
Se decidiamo di assegnare **TINYSEGM** a tutti i registri di segmento, possiamo informare l'assembler attraverso la direttiva **ASSUME**; come gia' sappiamo, questa direttiva serve per indicare all'assembler quale registro di segmento vogliamo utilizzare per referenziare un determinato segmento di programma. Nel programma di Figura 5 notiamo appunto una serie di direttive **ASSUME** che associano **TINYSEGM** a **CS**, **DS**, **ES** e **SS**; in questo modo stiamo dicendo all'assembler che tutti gli indirizzamenti che coinvolgono informazioni (codice, dati o stack) definite in **TINYSEGM** possono essere calcolati indifferentemente rispetto a **CS**, **DS**, **ES** o **SS**.
Quando l'assembler utilizza le varie direttive **ASSUME**, presuppone che a run time i registri **CS**, **DS**, **ES** e **SS** verranno materialmente inizializzati con il paragrafo di memoria che il **SO** assegna a **TINYSEGM**; in un precedente capitolo abbiamo visto che questo lavoro di inizializzazione spetta in parte al **SO** e in parte al programmatore.
E' necessario ricordare in particolare che il programmatore non deve inizializzare **CS** in quanto questo lavoro spetta al **SO**; infatti, al momento di caricare il programma in memoria, il **SO** carica in **CS** il paragrafo di memoria relativo al segmento di programma che contiene l'entry point. Inoltre il **SO** carica in **IP** l'offset dello stesso entry point; nel caso di Figura 5, in **CS** viene caricato **TINYSEGM**, e in **IP** l'offset dell'etichetta **start**.
L'inizializzazione dello stack invece e' necessaria in quanto nel nostro programma non e' presente

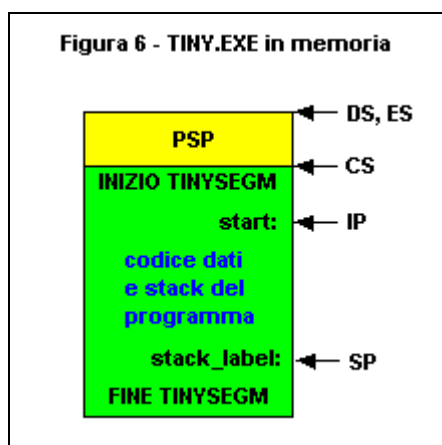
nessun segmento con attributo di combinazione **STACK**; come gia' sappiamo, se il **SO** trova un segmento di programma con attributo di combinazione **STACK**, assegna ad **SS** il corrispondente paragrafo di memoria, e posiziona **SP** alla fine dello stesso segmento di programma. Nell'esempio di Figura 5 viene mostrata una delle tante possibilita' che ci permettono di inizializzare correttamente lo stack quando questo lavoro e' a nostro carico; come si puo' notare, alla fine del programma viene creato un vettore **vstack** di **1024** byte, seguito da una etichetta **stack_label**. Grazie alla direttiva:

```
ALIGN 2,
```

si garantisce che questo vettore venga allineato alla **WORD** in modo da massimizzare le prestazioni dello stack; bisogna ricordare inoltre che l'area riservata allo stack deve essere formata da un numero pari di byte in modo che **SP** si venga a trovare sempre allineato ad offset multipli di **2**. Nel caso di Figura 5, lo stack pointer **SP** viene inizializzato con l'offset dell'etichetta **stack_label**, e quindi punta all'indirizzo (pari) immediatamente successivo a quello dell'ultimo elemento del vettore. Potevamo anche scrivere:

```
mov sp, offset vstack + STACK_SIZE.
```

In definitiva il nostro programma ha a disposizione **1024** byte di stack, localizzati alla estremita' finale del segmento **TINYSEGM**; la Figura 5 mette anche in rilievo il fatto che come gia' sappiamo, e' importantissimo inizializzare lo stack con le interruzioni hardware disabilitate.

Se ora proviamo ad assemblare il programma di Figura 5, richiedendo la generazione di un eseguibile in formato **EXE**, otteniamo un messaggio di avvertimento (**warning**) del linker che ci informa sul fatto che non e' presente nessuno stack; questo messaggio puo' essere tranquillamente ignorato in quanto e' dovuto al fatto che, come e' stato spiegato in precedenza, non abbiamo definito nessun segmento di programma con attributo di combinazione **STACK**.
Come gia' sappiamo, al momento di caricare in memoria un programma in formato **EXE**, il **SO** riserva un blocco di memoria capace di contenere non solo il programma stesso, ma anche **256** ulteriori byte destinati ad ospitare il **PSP** (Program Segment Prefix); il **PSP** viene sistemato nei primi **256** byte del blocco di memoria, ed e' immediatamente seguito dal programma. Successivamente il **SO** posiziona **DS** ed **ES** all'inizio del **PSP**, **CS** all'inizio di **TINYSEGM** e fa puntare **IP** all'etichetta **start**; appena caricato in memoria il programma **TINY.EXE** assume quindi l'aspetto mostrato in Figura 6 (in questa figura e in tutte quelle che seguono, gli indirizzi di memoria crescono dall'alto verso il basso).

A questo punto il controllo passa al programmatore che subito dopo l'entry point (**start**), deve provvedere innanzi tutto ad inizializzare i vari registri di segmento di sua competenza; l'esempio di Figura 5 mostra appunto il procedimento che porta all'inizializzazione di **DS**, **ES** e **SS**. Il listato di Figura 5 mostra anche le aree piu' adatte all'inserimento dei dati statici e delle procedure del programma; naturalmente il programmatore e' libero di complicarsi la vita trovando soluzioni piu'

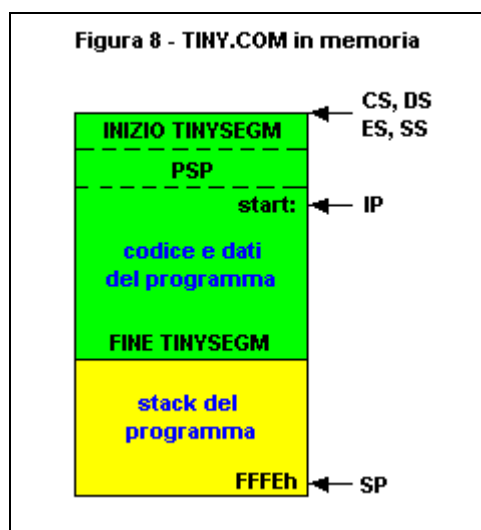
contorte di quelle illustrate nell'esempio. L'importante e' che il programma venga strutturato in modo da consentire alla CPU di poter distinguere chiaramente tra istruzioni principali, istruzioni delle procedure, dati statici e dati temporanei (stack) del programma.

Nel caso particolare dei programmi monosegmento, posssiamo ottenere notevoli semplificazioni strutturali richiedendo al linker la generazione di un eseguibile in formato **COM** (COre iMage); convertendo ad esempio in formato **COM** il programma di Figura 5, otteniamo la situazione mostrata in Figura 7.

Figura 7 - Modello di memoria TINY (COM)

```
;---------------------------------------------------;
; File tiny.asm                                     ;
; programma Assembly con modello di memoria TINY    ;
; (formato COM)                                     ;
;---------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

;################ segmento unico #################

TINYSEGM    SEGMENT  PARA PUBLIC USE16 'SEG_UNICO'

   assume   cs: TINYSEGM, ds: TINYSEGM
   assume   es: TINYSEGM, ss: TINYSEGM

   org      0100h                   ; 256 byte per il PSP

start:                              ; entry point

   jmp      start_code              ; salta le definizioni

;---------- inizio definizione variabili ----------

stringa1    db    'Chiamata NEAR di printStr1', 10, 13, '$'
stringa2    db    'Chiamata FAR di printStr2', 10, 13, '$'

;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

; void printStr1(char *strOffs)

printStr1 proc near

   strOffs  equ   [bp+4]            ; parametro strOffs

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp

   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      bp                      ; ripristina bp
   ret                              ; near return
```

```
printStr1 endp

; void far printStr2(char far *strAddr)
; strAddr = strSeg + strOffs

printStr2 proc far

   strOffs  equ   [bp+6]            ; parametro strOffs
   strSeg   equ   [bp+8]            ; parametro strSeg

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp
   push     ds                      ; preserva ds

   mov      ax, strSeg              ; carica strSeg
   mov      ds, ax                  ; in ds
   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      ds                      ; ripristina ds
   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr2 endp

;-------------- fine blocco procedure -------------

start_code:                         ; inizio codice principale

;------- inizio blocco principale istruzioni ------

   push     offset stringa1         ; parametro strOffs
   call     printStr1               ; chiama printStr1 (NEAR)
   add      sp, 2                   ; pulizia stack

   push     ds                      ; parametro strSeg
   push     offset stringa2         ; parametro strOffs
   call     printStr2               ; chiama printStr2 (FAR)
   add      sp, 4                   ; pulizia stack

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; servizio Return to DOS
   int      21h                     ; chiama i servizi DOS

;---------- inizio definizione variabili ----------

;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

;-------------- fine blocco procedure -------------

TINYSEGM    ENDS

;##################################################

   END      start
```

All'interno del segmento **TINYSEGM** notiamo subito la presenza della direttiva:
`ORG 0100h`,
che fa avanzare il **location counter** (**$**) di **256** byte; come gia' sappiamo, questi primi **256** byte del segmento **TINYSEGM** verranno riempiti dal **SO** con il **PSP** (Program Segment Prefix).
Ricordiamo inoltre che l'entry point del programma (etichetta **start**), deve trovarsi tassativamente a **256** byte di distanza dall'inizio del segmento unico di programma; e' proibito inserire qualsiasi informazione (codice o dati) prima dell'entry point. Notiamo anche che all'interno del programma e' scomparso qualsiasi riferimento al nome **TINYSEGM** (a run time); infatti, come abbiamo gia' visto nel Capitolo 13, in un programma in formato **COM** tutte le inizializzazioni dei registri di segmento spettano al **SO**, e qualsiasi riferimento a dati rilocabili (come **TINYSEGM**) provoca un messaggio di errore del linker.
Indipendentemente dalle dimensioni di **TINY.COM**, il **SO** riserva ad esso un blocco di memoria da **65536** byte; all'inizio di questo blocco viene sistemato il programma vero e proprio, comprendente anche il **PSP**. Tutti i byte che restano, cioe' tutto lo spazio compreso tra la fine di **TINYSEGM** e la fine del blocco di memoria (all'offset **FFFFh**), viene riservato allo stack; il **SO** posiziona poi **CS**, **DS**, **ES** e **SS** all'inizio di **TINYSEGM** (cioe' all'inizio del **PSP**), e fa puntare **IP** all'etichetta **start** e **SP** all'offset **FFFEh** (ultimo offset pari). Tutte queste inizializzazioni effettuate dal **SO** fanno risparmiare un bel po' di lavoro al programmatore; nella sezione **Assembly Avanzato** viene illustrato il metodo che bisogna seguire per ridurre al minimo le dimensioni di un eseguibile in formato **COM**. La Figura 8 mostra la situazione che si viene a creare non appena **TINY.COM** viene caricato in memoria.

Tornando alla Figura 7, possiamo vedere come ci si deve comportare se si vogliono inserire definizioni di dati e procedure prima del blocco di codice principale di un programma in formato **COM**; in pratica, subito dopo l'entry point (**start**), inseriamo una istruzione di salto incondizionato che fa in modo che la CPU scavalchi tutto il blocco delle definizioni. In questo modo si evita che la CPU si venga a trovare inavvertitamente nel bel mezzo di dati e procedure; naturalmente e' anche possibile inserire le definizioni di dati e procedure subito dopo le istruzioni di terminazione del programma, come viene indicato sempre dalla Figura 7.

Il pregio piu' importante di un programma con modello di memoria **TINY**, e' rappresentato dal fatto che tutti gli indirizzamenti che coinvolgono i dati, sono normalmente di tipo **NEAR** in quanto richiedono esclusivamente la componente **offset**; non e' necessario specificare la componente **seg** in quanto essa e' implicitamente rappresentata dal paragrafo di memoria assegnato all'unico segmento di programma presente. Qualunque procedura inoltre viene definita all'interno dello stesso (e unico) segmento di programma nel quale si trova anche il caller; questo significa che tutte le procedure del

programma possono essere definite di tipo **NEAR**. Per illustrare questi aspetti, vediamo che negli esempi di Figura 5 e di Figura 7 viene definita la procedura **printStr1** che e' di tipo **NEAR**; questa procedura richiede l'indirizzo **NEAR** di una stringa **DOS** che viene poi visualizzata con l'ausilio del servizio **Display String** dell'**INT 21h**. Il servizio **Display String** si aspetta una stringa puntata dalla coppia **DS:DX**; la procedura **printStr1** carica in **DX** l'offset della stringa da visualizzare e lascia inalterato **DS** in quanto questo registro referenzia gia' il segmento **TINYSEGM** nel quale e' stata definita la stringa stessa.
Naturalmente nessuno ci impedisce di definire procedure **FAR** o di utilizzare indirizzi **FAR** all'interno di un programma monosegmento; a tale proposito vediamo che negli esempi di Figura 5 e di Figura 7 viene definita la procedura **printStr2** che e' di tipo **FAR** e che richiede quindi una **FAR** call anche se il caller si trova all'interno dello stesso segmento di programma del called. La procedura **printStr2** richiede l'indirizzo **FAR** (cioe' una coppia **seg:offset**) di una stringa **DOS** che viene poi visualizzata con l'ausilio del servizio **Display String** dell'**INT 21h**; in questo caso la procedura **printStr2** prima di chiamare il servizio **Display String** pone **DS=seg** e **DX=offset**. Appare chiaro che la modifica di **DS** e' del tutto superflua in quanto **DS** referenzia gia' il segmento di programma nel quale e' stata definita la stringa (**DS=seg=TINYSEGM**); in ogni caso, questi esempi servono per illustrare il fatto che anche in un programma monosegmento possiamo lavorare se necessario con procedure **FAR** e con indirizzi **FAR** per i dati. E' importante sottolineare il fatto che una procedura che modifica **DS** ne deve preservare sempre il contenuto originario; questo aspetto diventa fondamentale nel caso dei programmi multisegmento che vengono illustrati piu' avanti.
Entrambe le procedure degli esempi di Figura 5 e di Figura 7 utilizzano le convenzioni **C** per il passaggio degli argomenti; indipendentemente da queste convenzioni, si deve tener presente che un indirizzo **FAR**, cioe' una coppia **seg:offset**, deve essere trattato come un unico argomento formato da **16+16** bit. Abbiamo visto che la convenzione seguita dalle CPU della famiglia **80x86** prevede che la componente **offset** di un indirizzo **FAR** occupi sempre i **16** bit meno significativi dell'indirizzo stesso; di conseguenza, se vogliamo passare un indirizzo **FAR** ad una procedura, dobbiamo inserire nello stack prima la componente **seg** e poi la componente **offset**. In questo modo le due componenti vengono sistemate in un blocco da **16+16** bit dello stack, con la componente **offset** che occcupa i **16** bit meno significativi; del resto questo e' lo stesso sistema che utilizza la CPU quando salva nello stack l'indirizzo di ritorno da una procedura.
Un ultimo aspetto importante riguarda il fatto che nel caso dell'esempio **TINY.COM** di Figura 7, sono state eliminate tutte le istruzioni che coinvolgono dati rilocabili come ad esempio:
`push seg stringa2`;
questa istruzione provoca un errore del linker in quanto l'operatore **SEG** dell'Assembly fa riferimento al segmento di appartenenza di **stringa2** che e' un valore rilocabile (**TINYSEGM**). Si tenga presente che il **MASM**, in relazione ai programmi in formato **COM**, non consente l'uso dell'operatore **FAR PTR** per la chiamata di una procedura **FAR**; per aggirare questo problema e' sufficiente definire le procedure prima della loro chiamata, cioe' prima del blocco di codice principale.

Il modello di memoria SMALL.

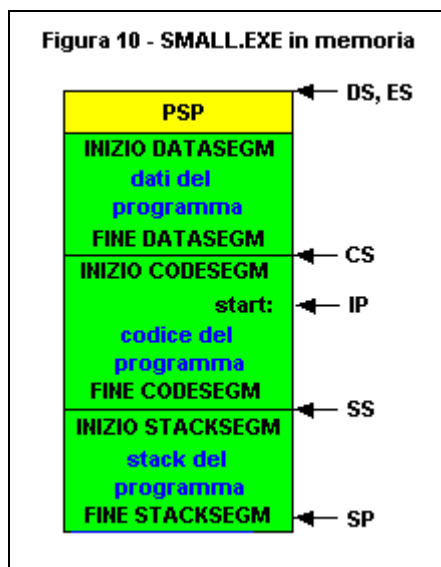
Considerando il fatto che le informazioni contenute in un programma Assembly possono essere suddivise in **codice**, **dati statici** e **dati temporanei** (stack), possiamo semplificarci la vita ripartendo queste informazioni in tre appositi segmenti di programma; a tale proposito possiamo definire un segmento destinato a contenere il codice, un segmento destinato a contenere i dati statici e un segmento destinato a contenere i dati temporanei (stack). Nella terminologia dei linguaggi di alto livello, un programma avente queste caratteristiche viene definito **programma con modello di memoria SMALL**; il termine **SMALL** in inglese significa piccolo. Naturalmente, nessuno ci impedisce eventualmente di mischiare codice, dati e stack all'interno di ciascuno di questi tre segmenti; la Figura 9 illustra un esempio nel quale possiamo notare la presenza di dati statici definiti in tutti i tre segmenti di programma.

Figura 9 - Modello di memoria SMALL

```
;--------------------------------------------------;
; File small.asm                                   ;
; Programma Assembly con modello di memoria SMALL  ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                              ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa1    db    'Stringa definita in DATASEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM    ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM          ; assegna CODESEGM a CS

;---------- inizio definizione variabili ----------

stringa2    db    'Stringa definita in CODESEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

; void printStr1(char *strOffs)

printStr1 proc near

   strOffs  equ   [bp+4]            ; parametro strOffs
```

```
   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp

   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr1 endp

; void printStr2(char far *strAddr)
; strAddr = strSeg + strOffs

printStr2 proc near

   strOffs  equu   [bp+4]           ; parametro strOffs
   strSeg   equ   [bp+6]            ; parametro strSeg

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp
   push     ds                      ; preserva ds

   mov      ax, strSeg              ; carica strSeg
   mov      ds, ax                  ; in ds
   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      ds                      ; ripristina ds
   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr2 endp

;-------------- fine blocco procedure -------------

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   push     offset stringa1         ; parametro strOffs
   call     near ptr printStr1      ; chiama printStr1 (NEAR)
   add      sp, 2                   ; pulizia stack

   push     seg stringa2            ; parametro strSeg
   push     offset stringa2         ; parametro strOffs
   call     near ptr printStr2      ; chiama printStr2 (NEAR)
   add      sp, 4

   push     seg stringa3            ; parametro strSeg
   push     offset stringa3         ; parametro strOffs
   call     near ptr printStr2      ; chiama printStr2 (NEAR)
   add      sp, 4

;-------- fine blocco principale istruzioni -------
```

```
   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; servizio Return to DOS
   int      21h                     ; chiama i servizi DOS

;------------- inizio blocco procedure ------------

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

;---------- inizio definizione variabili ----------

stringa3    db    'Stringa definita in STACKSEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

   align    2                             ; allineamento alla word

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

Nell'esempio di Figura 9 viene definito un segmento di programma **DATASEGM** destinato a contenere principalmente i dati statici, e un segmento **CODESEGM** destinato a contenere principalmente le istruzioni (comprese le procedure) del programma; viene inoltre definito un segmento **STACKSEGM** destinato a contenere principalmente lo stack del programma.
In un programma organizzato in questo modo, il programmatore deve preoccuparsi principalmente della opportuna inizializzazione di **DS** ed eventualmente di **ES**; l'inizializzazione di **CS** e di **SS** spetta invece al **SO**. Osserviamo infatti che l'entry point del programma (**start**) si trova nel blocco **CODESEGM**; di conseguenza il **SO** pone **CS=CODESEGM** e fa puntare **IP** all'etichetta **start**. Osserviamo inoltre che il blocco **STACKSEGM** e' dotato di attributo di combinazione **STACK**; di conseguenza il **SO** pone **SS=STACKSEGM** e fa puntare **SP** alla fine di **STACKSEGM**. E' imporatante ribadire che l'area riservata allo stack deve essere allineata ad indirizzi pari della memoria e deve avere una lunghezza pari in byte; per un segmento come **STACKSEGM** e' vivamente consigliato l'allineamento al paragrafo (**PARA**).
Naturalmente, quando si ha a che fare con programmi dotati di due o piu' segmenti distinti, e' possibile richiedere al linker la generazione di un eseguibile esclusivamente in formato **EXE**; al momento di caricare **SMALL.EXE** in memoria, il **SO** pone **DS=ES=PSP** e ottiene quindi la situazione mostrata in Figura 10.

A questo punto il controllo passa al programmatore che subito dopo l'entry point (**start**) deve provvedere a caricare materialmente in **DS** il segmento di programma che intende usare come segmento dati principale; nell'esempio di Figura 9 e' presente un solo segmento dati (**DATASEGM**), per cui la cosa piu' ovvia da fare consiste nel porre **DS=DATASEGM**. Ponendo anche **ES=DATASEGM** possiamo evitare parecchi segment override; come al solito, questo aspetto vale principalmente per le istruzioni per la manipolazione delle stringhe.

Ciascuno dei tre segmenti di cui e' dotato il programma di Figura 9, puo' crescere sino a **65536** byte; complessivamente il programma puo' raggiungere la dimensione massima di:
`65536 * 3 = 196608 byte = 192 Kb.`
Si tratta chiaramente di una dimensione piu' che sufficiente per soddisfare le esigenze della stragrande maggioranza dei programmi Assembly; proprio per questo motivo il modello di memoria **SMALL** e' quello piu' utilizzato nello sviluppo dei programmi.

In relazione agli indirizzamenti, il caso piu' semplice per un programma con modello di memoria **SMALL** si presenta quando, come si vede in Figura 10, tutto il codice viene isolato nel blocco codice, tutti i dati statici vengono isolati nel blocco dati e tutti i dati temporanei (stack) vengono isolati nel blocco stack; in una situazione del genere e' possibile lavorare esclusivamente con indirizzi **NEAR** per i dati e con procedure di tipo **NEAR**.
Osserviamo infatti che tutti i dati si trovano nel segmento **DATASEGM** che e' referenziato da **DS**; di conseguenza ci basta specificare solo la componente **offset** di un dato, in quanto la componente predefinita **seg** usata dalla CPU e' implicitamente **DS=DATASEGM**. Tecnicamente si puo' dire che **DS** e' il registro di segmento **naturale** per i dati di un programma; nel caso di Figura 9 essendo **DS=DATASEGM**, possiamo anche dire che **DATASEGM** e' il segmento naturale per i dati.
Analogamente si puo' osservare che tutto il codice del programma (comprese le procedure e le relative istruzioni di chiamata), si trova nel segmento **CODESEGM** referenziato da **CS**; di conseguenza possiamo lavorare esclusivamente con procedure di tipo **NEAR** e quindi con chiamate di tipo **NEAR**. Tecnicamente si puo' dire che **CS** e' il registro di segmento **naturale** per il codice di un programma; nel caso di Figura 9 essendo **CS=CODESEGM**, possiamo anche dire che **CODESEGM** e' il segmento naturale per il codice.
Infine notiamo che tutte le variabili temporanee del programma (stack) vengono isolate in un apposito segmento **STACKSEGM** referenziato da **SS**; in questo modo, tutte queste variabili possono essere indirizzate con la sola componente **offset** (**SP**, **BP**), in quanto la componente predefinita **seg** usata dalla CPU e' implicitamente **SS=STACKSEGM**. Tecnicamente si puo' dire

che **SS** e' il registro di segmento **naturale** per lo stack di un programma; nel caso di Figura 9 essendo **SS=STACKSEGM**, possiamo anche dire che **STACKSEGM** e' il segmento naturale per lo stack.
Per illustrare questi aspetti, vediamo che nell'esempio di Figura 9 viene definita una procedura **printStr1** di tipo **NEAR**; questa procedura richiede l'indirizzo **NEAR** di una stringa **DOS** che viene poi visualizzata attraverso il servizio **Display String** dell'**INT 21h**. Nel blocco di codice principale, a **printStr1** viene passata la componente **offset** della stringa **stringa1** definita in **DATASEGM**; la procedura **printStr1** pone **DX=offset** e chiama **Display String** che si aspetta una stringa **DOS** puntata da **DS:DX**. In pratica **printStr1** non modifica **DS** presupponendo che **DS=DATASEGM**; se invece **DS** punta da qualche altra parte, il comportamento di **Display String** diventa indeterminato, e puo' anche mandare in crash il programma.
Se la stringa da visualizzare viene definita in un segmento diverso da quello naturale, bisogna ricorrere agli indirizzi **FAR**; in Figura 9 notiamo la presenza della stringa **stringa2** definita nel blocco **CODESEGM** e della stringa **stringa3** definita nel blocco **STACKSEGM**. Come e' stato spiegato in un precedente capitolo, tenendo presente che lo stack inizia a riempirsi dagli indirizzi piu' alti, e' importante che le definizioni di eventuali dati nel blocco **STACKSEGM** vengano sistemate all'inizio del blocco stesso; in questo modo si riduce il rischio che **SP** decrementandosi possa sovrascrivere questi dati.
Per visualizzare queste stringhe **DOS** con il servizio **Display String**, utilizziamo la procedura **printStr2**, sempre di tipo **NEAR**; questa procedura richiede l'indirizzo completo **seg:offset** della stringa da visualizzare. In possesso della coppia **seg:offset** la procedura **printStr2** pone **DS=seg**, **DX=offset** e chiama **Dispaly String**; come si puo' notare **printStr2** preserva il contenuto originario del registro **DS**. Nel caso ad esempio di **stringa2**, osserviamo che inizialmente si ha **DS=DATASEGM**; la procedura **printStr2** riceve l'indirizzo completo **seg:offset** di **stringa2** con **seg=CODESEGM**. A questo punto **printStr2** pone **DS=seg=CODESEGM**, **DX=offset** e chiama **Display String**; prima di terminare **printStr** ripristina **DS** in modo da avere di nuovo **DS=DATASEGM**.

Gli esempi appena illustrati servono a sottolineare il fatto che finche' e' possibile, e' meglio sfruttare i segmenti di programma per isolare in modo chiaro e semplice il codice, i dati e lo stack; in questo modo possiamo evitare parecchie complicazioni legate in particolare all'uso degli indirizzi **FAR**. In ogni caso il programmatore Assembly deve essere in grado di padroneggiare tutte le situazioni possibili e immaginabili che si possono presentare; nel seguito del capitolo infatti, vengono presentati dei casi per i quali si rende obbligatorio il ricorso agli indirizzi **FAR** e/o alle procedure **FAR**.

Il modello di memoria MEDIUM.

Supponiamo di dover scrivere un programma che ha bisogno di meno di **64** Kb di dati, meno di **64** Kb di stack e di oltre **64** Kb di codice; in un caso del genere siamo costretti necessariamente a distribuire il codice su due o piu' segmenti di programma. La cosa migliore da fare consiste nel creare un programma dotato di un solo segmento di dati, un solo segmento di stack e di due o piu' segmenti di codice; nella terminologia dei linguaggi di alto livello un programma strutturato in questo modo viene definito **programma con modello di memoria MEDIUM** (medio). La Figura 11 illustra un esempio pratico.

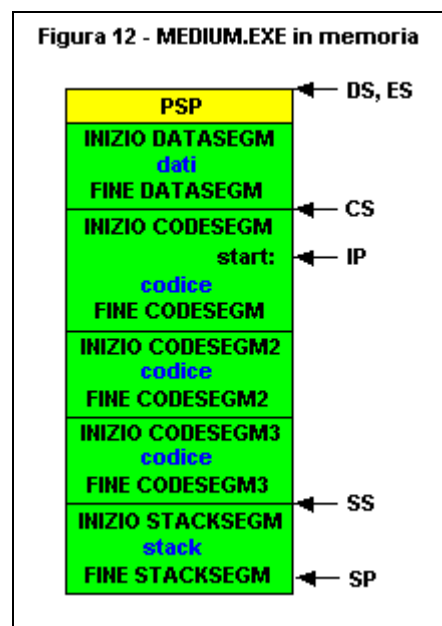
Figura 11 - Modello di memoria MEDIUM

```
;---------------------------------------------------;
; File medium.asm                                   ;
; Programma Assembly con modello di memoria MEDIUM ;
;---------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                  ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa1    db    'Stringa1 definita in DATASEGM', 10, 13, '$'
stringa2    db    'Stringa2 definita in DATASEGM', 10, 13, '$'
stringa3    db    'Stringa3 definita in DATASEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM    ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

;---------- inizio definizione variabili ----------

strCode1    db    'NEAR Call di printStr1 (CODESEGM)', 10, 13, '$'

;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

; void printStr1(char *strOffs)

printStr1 proc near

   strOffs  equ   [bp+4]            ; parametro strOffs

   push     bp                      ; preserva bp
```

```
   mov      bp, sp                 ; ss:bp = ss:sp

   push     ds                     ; preserva ds
   mov      ax, CODESEGM           ; trasferisce CODESEGM
   mov      ds, ax                 ; in ds
   mov      dx, offset strCode1    ; ds:dx punta a strCode1
   mov      ah, 09h                ; servizio Display String
   int      21h                    ; chiama i servizi DOS
   pop      ds                     ; ripristina ds

   mov      dx, strOffs            ; ds:dx punta alla stringa
   mov      ah, 09h                ; servizio Display String
   int      21h                    ; chiama i servizi DOS

   pop      bp                     ; ripristina bp
   ret                             ; near return

printStr1 endp

;-------------- fine blocco procedure -------------

start:                             ; entry point

   mov      ax, DATASEGM           ; trasferisce DATASEGM
   mov      ds, ax                 ; in DS attraverso AX
   assume   ds: DATASEGM           ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   push     offset stringa1        ; parametro strOffs
   call     near ptr printStr1     ; chiama printStr1 (NEAR)
   add      sp, 2                  ; pulizia stack

   push     offset stringa2        ; parametro strOffs
   call     far ptr printStr2      ; chiama printStr2 (FAR)
   add      sp, 2                  ; pulizia stack

   push     offset stringa3        ; parametro strOffs
   call     far ptr printStr3      ; chiama printStr3 (FAR)
   add      sp, 2                  ; pulizia stack

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                ; codice di uscita
   mov      ah, 4ch                ; servizio Return to DOS
   int      21h                    ; chiama i servizi DOS

;------------- inizio blocco procedure ------------

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;############## segmento codice 2 #################

CODESEGM2   SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM2          ; assegna CODESEGM2 a CS

;---------- inizio definizione variabili ----------

strCode2    db    'FAR Call di printStr2 (CODESEGM2)', 10, 13, '$'
```

```
;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

; void far printStr2(char *strOffs)

printStr2 proc far

   strOffs  equ   [bp+6]            ; parametro strOffs

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp

   push     ds                      ; preserva ds
   mov      ax, CODESEGM2           ; trasferisce CODESEGM2
   mov      ds, ax                  ; in ds
   mov      dx, offset strCode2     ; ds:dx punta a strCode2
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS
   pop      ds                      ; ripristina ds

   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr2 endp

;-------------- fine blocco procedure -------------

CODESEGM2   ENDS

;############## segmento codice 3 #################

CODESEGM3   SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM3           ; assegna CODESEGM3 a CS

;---------- inizio definizione variabili ----------

strCode3    db    'FAR Call di printStr3 (CODESEGM3)', 10, 13, '$'

;----------- fine definizione variabili -----------

;------------- inizio blocco procedure ------------

; void far printStr3(char *strOffs)

printStr3 proc far

   strOffs  equ   [bp+6]            ; parametro strOffs

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp

   push     ds                      ; preserva ds
   mov      ax, CODESEGM3           ; trasferisce CODESEGM3
   mov      ds, ax                  ; in ds
   mov      dx, offset strCode3     ; ds:dx punta a strCode3
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS
```

```
   pop      ds                       ; ripristina ds

   mov      dx, strOffs              ; ds:dx punta alla stringa
   mov      ah, 09h                  ; servizio Display String
   int      21h                      ; chiama i servizi DOS

   pop      bp                       ; ripristina bp
   ret                               ; near return

printStr3 endp

;-------------- fine blocco procedure -------------

CODESEGM3   ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT   PARA STACK USE16 'STACK'

            db        STACK_SIZE dup (?)    ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

E' importante ricordare che in un programma multisegmento, un solo segmento puo' contenere l'entry point; nel caso di Figura 11 vediamo che l'entry point, rappresentato dall'etichetta **start**, e' presente nel blocco **CODESEGM**. Di conseguenza il **SO** pone **CS=CODESEGM** e fa puntare **IP** all'offset dell'etichetta **start**; inoltre, in presenza di un blocco **STACKSEGM** con attributo di combinazione **STACK**, il **SO** pone **SS=STACKSEGM** e fa puntare **SP** alla fine di **STACKSEGM**. Infine, il **SO** pone **DS=PSP** e **ES=PSP**; non appena viene caricato in memoria, il programma **MEDIUM.EXE** assume quindi l'aspetto mostrato in Figura 12.

A questo punto il controllo passa al programmatore che deve preoccuparsi innanzi tutto della corretta inizializzazione di **DS**; essendoci un solo blocco dati che nel nostro caso si chiama **DATASEGM**, questa inizializzazione consiste nel porre **DS=DATASEGM**.
Per un programma con modello di memoria **MEDIUM** il caso piu' semplice si presenta come al solito quando le varie informazioni vengono inserite nei loro segmenti naturali; questo accade quando l'unico segmento di dati contiene solo dati statici, quando l'unico segmento di stack contiene solo dati temporanei e quando i vari segmenti di codice contengono solo codice. In un caso del genere si vede subito che per la gestione dei dati possiamo lavorare esclusivamente con indirizzi **NEAR**; essendoci infatti un solo segmento dati, non abbiamo bisogno di modificare **DS**. Per quanto riguarda le procedure invece, a causa della presenza di due o piu' segmenti di codice, siamo costretti a lavorare con chiamate di tipo **FAR**; questa situazione si presenta quando il caller e il called si trovano in due diversi segmenti di programma.
Per illustrare tutti questi aspetti possiamo fare riferimento all'esempio di Figura 11; in questo esempio notiamo la presenza di tre stringhe **stringa1**, **stringa2** e **stringa3**, che essendo definite nel segmento naturale **DATASEGM**, possono essere gestite con indirizzi **NEAR**. Infatti, queste tre stringhe vengono passate attraverso il loro solo offset, a tre procedure definite in tre segmenti di codice differenti; le tre procedure **printStr1**, **printStr2** e **printStr3** visualizzano queste tre stringhe presupponendo che **DS=DATASEGM**.
Nel caso in cui si vogliano utilizzare dati definiti in segmenti non naturali, bisogna ricorrere ad indirizzi di tipo **FAR**; a tale proposito, osserviamo che nell'esempio di Figura 11 sono presenti altre tre stringhe **strCode1**, **strCode2** e **strCode3**, definite rispettivamente in **CODESEGM**, **CODESEGM2** e **CODESEGM3**. Per poter visualizzare queste tre stringhe, le tre procedure **printStr1**, **printStr2** e **printStr3** ne devono conoscere l'indirizzo completo **seg:offset**; osserviamo che ad esempio **printStr3** per poter visualizzare **strCode3** con il servizio **Display String**, deve porre **DS=CODESEGM3**, e deve poi caricare in **DX** l'offset di **strCode3**. Come al solito, le procedure che modificano **DS** ne devono preservare il contenuto originario; nel caso delle tre procedure di Figura 11, se non viene ripristinato **DS**, il tentativo di visualizzare la stringa ricevuta come argomento (**strOffs**) fallisce.
Un altro aspetto da notare riguarda il fatto che **printStr1** e' di tipo **NEAR**; questo e' possibile in quanto questa procedura si trova in **CODESEGM** e viene chiamata dal blocco codice principale che si trova anch'esso in **CODESEGM**. Le altre due procedure sono invece di tipo **FAR** in quanto si trovano in segmenti diversi da quello del caller; di conseguenza, **printStr1** puo' chiamare **printStr2** o **printStr3** (con una **FAR** call), mentre **printStr2** e **printStr3** non possono chiamare **printStr1**.

Il programma dell'esempio di Figura 11 dispone di un segmento di dati, un segmento di stack e tre segmenti di codice; ciascuno di questi cinque segmenti puo' crescere sino a **65536** byte.
Complessivamente il programma puo' raggiungere la dimensione massima di:

```
65536 * 5 = 327680 byte = 320 Kb.
```

Al momento di eseguire questo programma, il **DOS** verifica se in memoria c'e' spazio a sufficienza; se questo spazio non e' disponibile viene mostrato un messaggio di errore che in genere informa l'utente che il programma e' troppo grande.
Come e' stato spiegato in un precedente capitolo, i programmi **DOS** hanno a disposizione circa **640** Kb di memoria **RAM** (convenzionale); per verificare la quantita' di memoria libera si puo' utilizzare il comando **mem /c** dal prompt del **DOS**.

Il modello di memoria COMPACT.

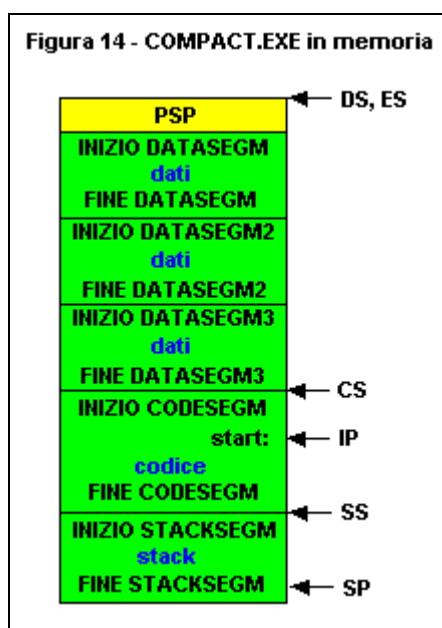
La situazione opposta rispetto a quella appena descritta si presenta quando dobbiamo scrivere un programma che ha bisogno di meno di **64** Kb di codice, meno di **64** Kb di stack e di oltre **64** Kb di dati; in un caso del genere siamo costretti necessariamente a distribuire i dati su due o piu' segmenti di programma. La cosa migliore da fare consiste nel creare un programma dotato di un solo segmento di codice, un solo segmento di stack e di due o piu' segmenti di dati; nella terminologia dei linguaggi di alto livello un programma strutturato in questo modo viene definito **programma con modello di memoria COMPACT** (compatto). La Figura 13 illustra un esempio pratico.

Figura 13 - Modello di memoria COMPACT

```
;---------------------------------------------------;
; File compact.asm                                  ;
; Programma Assembly con modello di memoria COMPACT ;
;---------------------------------------------------;

;########### direttive per l'assembler ############

.386                              ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa1    db    'Stringa1 definita in DATASEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM    ENDS

;############### segmento dati 2 ##################

DATASEGM2   SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa2    db    'Stringa2 definita in DATASEGM2', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM2   ENDS

;############### segmento dati 3 ##################

DATASEGM3   SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa3    db    'Stringa3 definita in DATASEGM3', 10, 13, '$'

;----------- fine definizione variabili -----------
```

```
DATASEGM3   ENDS

;############### segmento codice #################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

;------------- inizio blocco procedure ------------

; void printStr1(char far *strAddr)
; strAddr = strSeg + strOffs

printStr1 proc near

   strOffs  equ   [bp+4]            ; parametro strOffs
   strSeg   equ   [bp+6]            ; parametro strSeg

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp
   push     ds                      ; preserva ds

   mov      ds, strSeg              ; ds = seg stringa
   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      ds                      ; ripristina ds
   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr1 endp

;-------------- fine blocco procedure -------------

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   push     seg stringa1            ; parametro strSeg
   push     offset stringa1         ; parametro strOffs
   call     near ptr printStr1      ; chiama printStr1 (NEAR)
   add      sp, 4                   ; pulizia stack

   push     seg stringa2            ; parametro strSeg
   push     offset stringa2         ; parametro strOffs
   call     near ptr printStr1      ; chiama printStr1 (NEAR)
   add      sp, 4                   ; pulizia stack

   push     seg stringa3            ; parametro strSeg
   push     offset stringa3         ; parametro strOffs
   call     near ptr printStr1      ; chiama printStr1 (NEAR)
   add      sp, 4                   ; pulizia stack

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; servizio Return to DOS
   int      21h                     ; chiama i servizi DOS
```

```
;------------- inizio blocco procedure ------------

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;################ segmento stack ################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

In presenza dell'unico segmento di codice **CODESEGM**, contenente l'entry point **start**, il **SO** pone **CS=CODESEGM** e fa puntare **IP** a **start**; analogamente, in presenza del blocco **STACKSEGM** con attributo di combinazione **STACK**, il **SO** pone **SS=STACKSEGM**, e fa puntare **SP** alla fine di **STACKSEGM**. Infine il **SO** pone **DS=PSP** e **ES=PSP**; non appena viene caricato in memoria, il programma **COMPACT.EXE** assume quindi l'aspetto mostrato in Figura 14.

A questo punto il controllo passa al programmatore che deve preoccuparsi innanzi tutto della corretta inizializzazione di **DS**; nel caso del nostro esempio notiamo la presenza di tre segmenti di dati, per cui dobbiamo decidere quale di essi deve essere utilizzato per primo. In genere, uno dei segmenti di dati assume il ruolo di segmento dati principale; questo segmento puo' essere utilizzato per inizializzare **DS**. Un altro segmento dati puo' essere usato per inizializzare **ES**; abilitando inoltre il set di istruzioni a **32** bit e' possibile sfruttare anche gli altri due registri di segmento **FS** e **GS** per i dati.

Nel caso dei tre segmenti di dati di Figura 14, volendo utilizzare ad esempio **DATASEGM** come segmento dati principale, possiamo scrivere:

```
mov      ax, DATASEGM      ; trasferisce DATASEGM
mov      ds, ax            ; in DS attraverso AX
assume   ds: DATASEGM      ; assegna DATASEGM a DS

mov      ax, DATASEGM2     ; trasferisce DATASEGM2
mov      es, ax            ; in ES attraverso AX
assume   es: DATASEGM      ; assegna DATASEGM2 a ES

mov      ax, DATASEGM3     ; trasferisce DATASEGM3
mov      fs, ax            ; in FS attraverso AX
assume   fs: DATASEGM3     ; assegna DATASEGM3 a FS
```

Le direttive **ASSUME** sono importanti solo se vogliamo accedere per nome alle variabili definite nei vari segmenti di dati, e vogliamo delegare all'assembler il compito di inserire i vari segment override; se non usiamo le direttive **ASSUME**, tutti i segment override sono a nostro carico. E' chiaro che se il nostro programma e' dotato di **5** o piu' segmenti di dati, i quattro registri **DS**, **ES**, **FS** e **GS** non sono piu' sufficienti; in questo caso, possiamo usare uno o piu' di questi registri per referenziare a turno i vari segmenti di dati ai quali vogliamo accedere.

Per un programma con modello di memoria **COMPACT** il caso piu' semplice si presenta come al solito quando le varie informazioni vengono inserite nei loro segmenti naturali; questo accade quando l'unico segmento di codice contiene solo codice, quando l'unico segmento di stack contiene solo dati temporanei e quando i vari segmenti di dati contengono solo dati. In un caso del genere, l'unico segmento di codice presente contiene sia le varie procedure del programma che le relative istruzioni di chiamata; di conseguenza possiamo lavorare esclusivamente con procedure di tipo **NEAR** che richiedono quindi **NEAR** calls. Per quanto riguarda i dati invece, a causa della presenza di due o piu' segmenti di dati, siamo costretti a lavorare con indirizzamenti di tipo **FAR**; in sostanza, quando vogliamo accedere ad una determinata variabile, dobbiamo specificare non solo il suo offset, ma anche il segmento di appartenenza.
Tutti questi concetti vengono illustrati nel programma di esempio di Figura 13; in questo esempio si nota la presenza di tre stringhe **stringa1**, **stringa2** e **stringa3**, definite rispettivamente nei tre segmenti di dati **DATASEGM**, **DATASEGM2** e **DATASEGM3**. Queste tre stringhe vengono passate per indirizzo ad una procedura **printStr1** che le visualizza con il servizio **Display String**; **printStr1** e' di tipo **NEAR** in quanto viene chiamata dal codice principale del programma, che si trova nello stesso blocco **CODESEGM** della procedura. Il programma di Figura 13 puo' fare a meno delle direttive **ASSUME** per i dati in quanto stiamo lavorando esclusivamente con gli indirizzi dei dati stessi; gli operatori dell'Assembly come **SEG** e **OFFSET** applicati ad una determinata variabile, agiscono direttamente sul segmento di appartenenza della variabile stessa.
Nel caso ad esempio di **stringa3**, l'istruzione:
`push seg stringa3,`
inserisce nello stack il valore **DATASEGM3**; analogamente, l'istruzione:
`push offset stringa3,`
inserisce nello stack il valore **0** che e' l'offset di **stringa3** calcolato rispetto a **DATASEGM3**.

Per quanto riguarda infine gli aspetti legati alla dimensione massima raggiungibile da un programma con modello di memoria **COMPACT**, valgono tutte le considerazioni gia' svolte per il modello di memoria **MEDIUM**.

Il modello di memoria LARGE.

In relazione alle esigenze di memoria di un programma, il caso piu' gravoso si presenta quando abbiamo bisogno di oltre **64** Kb di codice e di oltre **64** Kb di dati; in un caso del genere, sia il codice che i dati devono essere distribuiti su due o piu' segmenti di programma. Per quanto riguarda lo stack, e' piuttosto difficile che un programma abbia bisogno di oltre **64** Kb di dati temporanei; in una eventualita' del genere, la gestione di due o piu' segmenti di stack diventa particolarmente complessa. Nei **SO** multitasking capaci di far girare piu' programmi contemporaneamente, ogni programma in esecuzione e' dotato del proprio stack; questi aspetti vengono illustrati nella sezione **Modalita' Protetta**.
Nella terminologia dei linguaggi di alto livello, un programma dotato di piu' segmenti di codice, piu' segmenti di dati e un solo segmento di stack, viene definito **programma con modello di memoria LARGE** (grande); la Figura 15 illustra un esempio pratico.

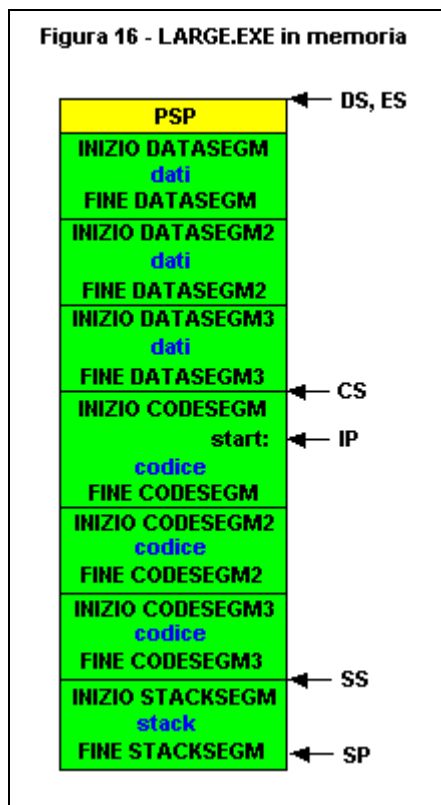
Figura 15 - Modello di memoria LARGE

```
;---------------------------------------------------;
; File large.asm                                    ;
; Programma Assembly con modello di memoria LARGE  ;
;---------------------------------------------------;

;########### direttive per l'assembler ############

.386                            ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h              ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa1    db    'Stringa1 definita in DATASEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM    ENDS

;############### segmento dati 2 ##################

DATASEGM2   SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa2    db    'Stringa2 definita in DATASEGM2', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM2   ENDS

;############### segmento dati 3 ##################

DATASEGM3   SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------
```

```
stringa3    db    'Stringa3 definita in DATASEGM3', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM3   ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

;------------- inizio blocco procedure ------------

; void printStr1(char far *strAddr)
; strAddr = strSeg + strOffs

printStr1 proc near

   strOffs  equu   [bp+4]            ; parametro strOffs
   strSeg   equ    [bp+6]            ; parametro strSeg

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:sp
   push     ds                      ; preserva ds

   mov      ds, strSeg              ; ds = seg stringa
   mov      dx, strOffs             ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      ds                      ; ripristina ds
   pop      bp                      ; ripristina bp
   ret                              ; near return

printStr1 endp

;-------------- fine blocco procedure -------------

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   push     seg stringa1            ; parametro strSeg
   push     offset stringa1         ; parametro strOffs
   call     near ptr printStr1      ; chiama printStr1 (NEAR)
   add      sp, 4                   ; pulizia stack

   push     seg stringa2            ; parametro strSeg
   push     offset stringa2         ; parametro strOffs
   call     far ptr printStr2       ; chiama printStr2 (FAR)
   add      sp, 4                   ; pulizia stack

   push     seg stringa3            ; parametro strSeg
   push     offset stringa3         ; parametro strOffs
   call     far ptr printStr3       ; chiama printStr3 (FAR)
   add      sp, 4                   ; pulizia stack

;-------- fine blocco principale istruzioni -------
```

```
   mov      al, 00h                ; codice di uscita
   mov      ah, 4ch                ; servizio Return to DOS
   int      21h                    ; chiama i servizi DOS

;------------- inizio blocco procedure ------------

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;############## segmento codice 2 #################

CODESEGM2   SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM2          ; assegna CODESEGM2 a CS

;------------- inizio blocco procedure ------------

; void far printStr2(char far *strAddr)
; strAddr = strSeg + strOffs

printStr2 proc far

   strOffs  equu   [bp+6]          ; parametro strOffs
   strSeg   equ   [bp+8]           ; parametro strSeg

   push     bp                     ; preserva bp
   mov      bp, sp                 ; ss:bp = ss:sp
   push     ds                     ; preserva ds

   mov      ds, strSeg             ; ds = seg stringa
   mov      dx, strOffs            ; ds:dx punta alla stringa
   mov      ah, 09h                ; servizio Display String
   int      21h                    ; chiama i servizi DOS

   pop      ds                     ; ripristina ds
   pop      bp                     ; ripristina bp
   ret                             ; near return

printStr2 endp

;-------------- fine blocco procedure -------------

CODESEGM2   ENDS

;############## segmento codice 3 #################

CODESEGM3   SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM3          ; assegna CODESEGM3 a CS

;------------- inizio blocco procedure ------------

; void far printStr3(char far *strAddr)
; strAddr = strSeg + strOffs

printStr3 proc far

   strOffs  equ   [bp+6]           ; parametro strOffs
   strSeg   equ   [bp+8]           ; parametro strSeg

   push     bp                     ; preserva bp
```

```
   mov      bp, sp                   ; ss:bp = ss:sp
   push     ds                       ; preserva ds

   mov      ds, strSeg               ; ds = seg stringa
   mov      dx, strOffs              ; ds:dx punta alla stringa
   mov      ah, 09h                  ; servizio Display String
   int      21h                      ; chiama i servizi DOS

   pop      ds                       ; ripristina ds
   pop      bp                       ; ripristina bp
   ret                               ; near return

printStr3 endp

;-------------- fine blocco procedure -------------

CODESEGM3   ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

Nell'esempio di Figura 15 si nota che l'entry point **start** e' stato sistemato all'interno del segmento **CODESEGM**; di conseguenza il **SO** pone **CS=CODESEGM** e fa puntare **IP** a **start**. Il segmento **STACKSEGM** e' dotato di attributo di combinazione **STACK**; di conseguenza il **SO** pone **SS=STACKSEGM** e fa puntare **SP** alla fine di **STACKSEGM**. Infine il **SO** pone **DS=PSP** e **ES=PSP**; non appena viene caricato in memoria, il programma **LARGE.EXE** assume quindi l'aspetto mostrato in Figura 16.

A questo punto il controllo passa al programmatore che deve preoccuparsi innanzi tutto della corretta inizializzazione di **DS**; per l'esempio di Figura 15, valgono le stesse considerazioni esposte per l'esempio di Figura 13, per cui trattando **DATASEGM** come blocco dati principale, poniamo **DS=DATASEGM**.
Nel caso di un programma con modello di memoria **LARGE**, siamo costretti a lavorare con indirizzamenti **FAR** per i dati e con procedure di tipo **FAR**; questa situazione e' dovuta al fatto che sia i dati che le procedure sono distribuiti su due o piu' segmenti di programma.
Il programma di esempio di Figura 15 ha il compito di illustrare in pratica questi aspetti; in questo esempio si nota la presenza di tre stringhe **stringa1**, **stringa2** e **stringa3**, definite rispettivamente nei tre segmenti di dati **DATASEGM**, **DATASEGM2** e **DATASEGM3**. Ciascuna di queste tre stringhe viene passata per indirizzo ad una apposita procedura che provvede a visualizzarla con il servizio **Display String**; osserviamo che **printStr1** e' di tipo **NEAR** in quanto viene chiamata dal codice principale del programma, che si trova nello stesso blocco **CODESEGM** della procedura.
Ciascuna delle tre procedure ha bisogno di conoscere l'indirizzo completo **seg:offset** della stringa da visualizzare; questo perche' le varie stringhe vengono definite in diversi segmenti di dati.

Il modello di memoria HUGE.

Alcuni compilatori di linguaggi come il **C/C++**, mettono a disposizione anche il modello di memoria **HUGE** (enorme); il modello **HUGE** ha la stessa struttura del modello **LARGE**, ed ha lo scopo di permettere al programmatore di definire singoli dati (generalmente grossi vettori) che superano la dimensione di **64** Kb. In precedenza e' stato detto che ogni segmento di programma con attributo **USE16** non puo' superare la dimensione di **64** Kb; questo significa che non e' possibile definire ad esempio il seguente vettore:

```
bigvect dw 40000 dup (0).
```

Osserviamo infatti che il vettore **bigvect** e' formato da **40000** word e richiede quindi complessivamente uno spazio di memoria pari a **80000** byte; questa dimensione supera quella massima consentita per un segmento di programma, e quindi si ottiene un messaggio di errore da parte dell'assembler.
Questa limitazione viene superata da alcuni compilatori attraverso l'uso di apposite procedure che permettono di gestire grossi dati via software; il trucco che viene utilizzato consiste nel sostituire gli indirizzi logici con gli **Indirizzi Logici Normalizzati** (in seguito viene utilizzata l'abbreviazione **ILN**). Per capire di cosa si tratta, riprendiamo l'esempio mostrato in Figura 3; in quell'esempio abbiamo visto che a causa della parziale sovrapposizione dei segmenti di memoria, uno stesso indirizzo fisico puo' essere rappresentato con due o piu' indirizzi logici. Consideriamo ad esempio l'indirizzo fisico a **20** bit **00037h** (**55d**); come si vede in Figura 3, questo indirizzo ricade nei primi **4** segmenti di memoria, per cui puo' essere rappresentato dai seguenti **4** indirizzi logici:

```
00037h = 0000h:0037h
00037h = 0001h:0027h
00037h = 0002h:0017h
00037h = 0003h:0007h
```

Tra gli indirizzi logici e gli indirizzi fisici non esiste quindi una corrispondenza biunivoca; questa ambiguita' impedisce ai compilatori **C/C++** di effettuare correttamente confronti tra indirizzi logici.
Osserviamo infatti che dal punto di vista del compilatore **C/C++**, i **4** indirizzi logici precedenti appaiono tutti distinti tra loro; in realta' sappiamo che questi **4** indirizzi logici rappresentano tutti l'indirizzo fisico **00037h**, e quindi sono tutti uguali tra loro.
In un precedente capitolo abbiamo anche visto che questo tipo di rappresentazione degli indirizzi logici, presenta un problema che viene chiamato **wrap around** (attorcigliamento); considerando ad esempio l'indirizzo logico:

```
0004h:FFFFh,
```

se proviamo a sommare **1** alla componente **offset** otteniamo:

```
FFFFh + 0001h = 10000h.
```

A causa del fatto che la componente **offset** e' un valore a **16** bit, la cifra piu' significativa di **10000h** viene persa, e quindi non si ottiene:

```
0005h:0000h,
```

ma bensi':

```
0004h:0000h.
```

In pratica, ogni volta che un offset supera **FFFFh**, ricomincia da **0000h** sempre all'interno dello stesso segmento di memoria; da queste considerazioni appare chiaro il fatto che il programmatore deve stare attento a non superare mai la barriera **FFFFh** per gli offset.
Il fenomeno del wrap around ci impedisce di fatto di gestire grossi vettori che superano la dimensione di **64** Kb; questo problema puo' essere aggirato con l'uso degli **ILN**. Un **ILN** e' un indirizzo logico che utilizza i soliti **16** bit per la componente **seg** e solamente **4** bit per la componente **offset**. La conversione di un indirizzo fisico in un **ILN** e' semplicissima; basta prendere l'indirizzo fisico a **20** bit, separare i **4** bit meno significativi dai restanti **16** bit e il gioco e' fatto; convertendo tutto in esadecimale possiamo dire che l'indirizzo fisico a **5** cifre viene suddiviso in due

parti separando la cifra meno significativa dalle restanti **4** cifre. Nel caso ad esempio dell'indirizzo fisico **3BF2Ah** si ottiene:
`3BF2Ah = 3BF2h:Ah.`
Per riconvertire un **ILN** in un indirizzo fisico si utilizza ovviamente la solita formula:
`indirizzo fisico = (seg * 16) + offset;`
Nel caso dell'**ILN 3BF2h:Ah** si ottiene:
`(3BF2h * 10h) + Ah = 3BF20h + Ah = 3BF2Ah.`
Si puo' facilmente constatare che questa volta, tra gli indirizzi fisici e gli **ILN** esiste una corrispondenza biunivoca; infatti, ad ogni indirizzo fisico corrisponde uno e un solo **ILN**, e viceversa, ad ogni **ILN** corrisponde uno e un solo indirizzo fisico. Nel caso ad esempio dell'indirizzo fisico **00037h** di Figura 3, si ottengono **4** indirizzi logici differenti, e un solo **ILN** che vale **0003h:7h**.
Con i **4** bit della componente **offset** di un **ILN** possiamo rappresentare tutti gli offset che vanno da **0h** a **Fh**; in pratica, la componente **offset** di un **ILN** rappresenta uno spiazzamento all'interno del paragrafo di memoria **seg**. Si puo' subito osservare che gli **ILN** eliminano il problema del wrap around; per dimostrarlo riprendiamo l'esempio relativo all'indirizzo logico:
`0004h:FFFFh.`
Prima di tutto convertiamo questo indirizzo logico nel corrispondente indirizzo fisico:
`0004h:FFFFh = 00040h + 0FFFFh = 1003Fh.`
Convertiamo ora questo indirizzo fisico nel corrispondente **ILN**:
`1003Fh = 1003h:Fh.`
Sommando **1** alla componente **offset** di questo **ILN** otteniamo l'indirizzo logico ordinario:
`1003h:(Fh + 1h) = 1003h:10h,`
che una volta normalizzato diventa:
`1004h:0h.`
Come si puo' notare, siamo passati da **1003h:Fh** a **1004h:0h**; in pratica siamo passati dall'ultimo byte del paragrafo **1003h** al primo byte del paragrafo **1004h**. Possiamo dire quindi che grazie agli **ILN**, possiamo scandire tutto il Mb di RAM dell'**8086** senza il problema del wrap around; il tutto naturalmente in modalita' di indirizzamento reale **8086**.

Nella terminologia dei linguaggi di alto livello, un dato di dimensioni superiori a **64** Kb viene chiamato **dato huge**; un esempio pratico e' rappresentato proprio dal vettore **bigvect** definito in precedenza. Per gestire dati **huge**, i compilatori **C/C++** li distribuiscono su due o piu' segmenti di dati; nel caso ad esempio del vettore **bigvect** possiamo avere la seguente situazione:

```
DATASEGM1   SEGMENT PARA PUBLIC USE16 'DATA'

bigvect     dw    30000 dup (0)

DATASEGM1   ENDS

DATASEGM2   SEGMENT WORD PUBLIC USE16 'DATA'

            dw    10000 dup (0)

DATASEGM2   ENDS
```

Il primo blocco **DATASEGM1** contiene le prime **30000** word di **bigvect**, pari a **60000** byte; notiamo che il blocco **DATASEGM1** e' allineato al paragrafo (**PARA**), per cui il primo elemento di **bigvect** si trova sicuramente all'offset **0000h**. Il secondo blocco **DATASEGM2** contiene le restanti **10000** word di **bigvect**, pari a **20000** byte; per fare i modo che le **40000** word di **bigvect** vengano disposte in memoria in locazioni consecutive e contigue, dobbiamo assegnare a **DATASEGM2** un attributo di allineamento di tipo **WORD**. Infatti, supponiamo che a **DATASEGM1** venga assegnato

il paragrafo di memoria **0100h** corrispondente all'indirizzo fisico **01000h**; siccome in questo blocco dati sono presenti **60000** byte (**EA60h**), possiamo dire che **DATASEGM1** parte dall'indirizzo logico **0100h:0000h**, e termina all'indirizzo logico **0100h:EA5Fh** (ricordiamo che gli indici partono da zero, per cui tra l'offset **0000h** e l'offset **EA5Fh** ci sono **EA60h** byte). Tutto cio' equivale a dire che **DATASEGM1** parte dall'indirizzo fisico **01000h**, e termina all'indirizzo fisico **0FA5Fh**; a questo punto il **SO** deve caricare in memoria il blocco **DATASEGM2** per il quale abbiamo richiesto un allineamento di tipo **WORD**. Siccome l'indirizzo fisico **0FA60h** (cioe' quello successivo a **0FA5Fh**) e' un numero pari, compatibile quindi con l'allineamento di tipo **WORD**, il **SO** fa partire **DATASEGM2** proprio da **0FA60h**; l'indirizzo fisico **0FA60h** equivale all'indirizzo logico **0FA6h:0000h**, e quindi a **DATASEGM2** viene assegnato il paragrafo di memoria **0FA6h**. Inoltre, si nota subito che il primo byte contenuto in **DATASEGM2** parte chiaramente dall'indirizzo fisico **0FA60h**, e quindi segue immediatamente l'ultimo byte di **DATASEGM1** che si trova all'indirizzo fisico **0FA5Fh**; siccome **DATASEGM2** contiene **20000** byte (**4E20h**), possiamo dire che questo blocco dati parte dall'indirizzo logico **0FA6h:0000h** e termina all'indirizzo logico **0FA6h:4E1Fh**. Tutto cio' equivale a dire che **DATASEGM2** parte dall'indirizzo fisico **0FA60h**, e termina all'indirizzo fisico **1487Fh**; complessivamente, gli **80000** byte di **bigvect** vengono disposti in memoria in modo consecutivo e contiguo dall'indirizzo fisico **01000h** all'indirizzo fisico **1487Fh**. Abbiamo infatti:
`1487Fh - 01000h = 1387Fh = 79999 byte`;
tenendo conto del byte di indice **0** che si trova all'indirizzo fisico **01000h**, otteniamo in totale **80000** byte.

Per gestire un dato **huge** come **bigvect**, i compilatori **C/C++** utilizzano gli **ILN**; in questo modo possono scandire tutto il vettore senza il rischio del wrap around. Nel caso del nostro esempio vediamo che all'interno di **DATASEGM1**, la prima word di **bigvect** si trova all'**ILN 0100h:0h**, la seconda word si trova a **0100h:2h**, la terza word si trova a **0100h:4h**, e cosi' via sino all'ultima word che si trova all'indirizzo fisico **0FA5Eh**, e cioe' all'**ILN 0FA5h:Eh**; all'interno di **DATASEGM2**, la prima word di **bigvect** si trova all'**ILN 0FA6h:0h**, la seconda word si trova a **0FA6h:2h**, la terza word si trova a **0FA6h:4h**, e cosi' via sino all'ultima word che si trova all'indirizzo fisico **1487Eh**, e cioe' all'**ILN 1487h:Eh**. Come si puo' notare, ogni volta che si supera l'offset **Fh** si passa al paragrafo successivo; in questo modo e' possibile distinguere in modo univoco tutte le **40000** word di **bigvect**. L'utilizzo degli **ILN** diventa fondamentale nel momento in cui si ha bisogno di confrontare tra loro due indirizzi relativi a due elementi di un dato **huge**; grazie al fatto che gli **ILN** sono tutti distinti tra loro, il compilatore **C/C++** e' in grado di applicare ad essi gli operatori relazionali come **<, >, ==, <=**, etc.
L'aspetto negativo degli **ILN** e' rappresentato dal fatto che il compilatore **C/C++** li gestisce via software attraverso apposite procedure; e' chiaro infatti che la CPU lavora via hardware con indirizzi logici ordinari e non con gli **ILN**. Di conseguenza, un programma **C/C++** che fa uso del modello di memoria **HUGE** e degli **ILN**, presenta maggiori dimensioni e minore velocita' rispetto ad un programma con modello di memoria **LARGE** che usa gli indirizzi **FAR**.

La direttiva GROUP.

In molti casi e' possibile semplificare la struttura di un programma, raggruppando tra loro due o piu' segmenti distinti; questa tecnica viene utilizzata ad esempio dai compilatori **C/C++** per raggruppare dati temporanei (stack) e diverse categorie di dati statici. Si tratta quindi di una caratteristica concepita espressamente per i compilatori dei linguaggi di alto livello; in generale un programmatore Assembly non ha bisogno di ricorrere ad un simile espediente. In ogni caso si tratta di aspetti che e' importante conoscere in quanto possono tornare utili quando si ha bisogno di interfacciare l'Assembly con i linguaggi di alto livello; per consentire il raggruppamento di due o piu' segmenti di programma, l'Assembly mette a disposizione la direttiva **GROUP**. Attraverso questa direttiva e' possibile utilizzare un nome unico per gestire i vari segmenti che fanno parte del gruppo; la Figura 17 mostra un esempio pratico che fa uso della libreria **EXELIB**.

Figura 17 - Uso della direttiva GROUP

```
;---------------------------------------------------;
; File group.asm                                    ;
; Programma Assembly con la direttiva GROUP         ;
;---------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit
INCLUDE     EXELIB.INC              ; inclusione libreria di I/O

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                  ; 1024 byte per lo stack

;############### segmento dati 1 ##################

DSEG1       SEGMENT  PARA PUBLIC USE16 'DATA'

            dw    0
varWord1    dw    31800
            dw    0
            dw    0

DSEG1       ENDS

;############### segmento dati 2 ##################

DSEG2       SEGMENT  PARA PUBLIC USE16 'DATA'

            dw    0
            dw    0
varWord2    dw    12900
            dw    0

DSEG2       ENDS

;############### segmento dati 3 ##################

DSEG3       SEGMENT  PARA PUBLIC USE16 'DATA'

            dw    0
            dw    0
            dw    0
varWord3    dw    11600
```

```
DSEG3       ENDS

;################ gruppo dgroup ##################

DGROUP      GROUP    DSEG1, DSEG2, DSEG3

;############### segmento codice #################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

   mov      ax, DGROUP              ; trasferisce DGROUP
   mov      es, ax                  ; in ES attraverso AX
   assume   es: DGROUP              ; assegna DGROUP a ES

;------- inizio blocco principale istruzioni ------

   call     clearScreen             ; pulizia schermo

; LEA

   mov      dx, 0400h               ; riga 04h, colonna 00h
   lea      ax, varWord1            ; ax = offset varWord1
   call     writeHex16              ; visualizza ax

   inc      dh                      ; incremento riga
   lea      ax, varWord2            ; ax = offset varWord2
   call     writeHex16              ; visualizza ax

   inc      dh                      ; incremento riga
   lea      ax, varWord3            ; ax = offset varWord3
   call     writeHex16              ; visualizza ax

; OFFSET

   add      dh, 2                   ; incremento riga
   mov      ax, offset varWord1     ; ax = offset varWord1
   call     writeHex16              ; visualizza ax

   inc      dh                      ; incremento riga
   mov      ax, offset varWord2     ; ax = offset varWord2
   call     writeHex16              ; visualizza ax

   inc      dh                      ; incremento riga
   mov      ax, offset varWord3     ; ax = offset varWord3
   call     writeHex16              ; visualizza ax

; OFFSET + segment override

   add      dh, 2                      ; incremento riga
   mov      ax, offset DGROUP:varWord1 ; ax = offset varWord1
   call     writeHex16                 ; visualizza ax

   inc      dh                         ; incremento riga
   mov      ax, offset DGROUP:varWord2 ; ax = offset varWord2
```

```
   call     writeHex16                ; visualizza ax

   inc      dh                        ; incremento riga
   mov      ax, offset DGROUP:varWord3 ; ax = offset varWord3
   call     writeHex16                ; visualizza ax


;-------- fine blocco principale istruzioni -------

   mov      al, 00h                ; codice di uscita
   mov      ah, 4ch                ; servizio Return to DOS
   int      21h                    ; chiama i servizi DOS

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;###################################################

   END      start
```

Nel programma **GROUP.ASM** di Figura 17 notiamo la presenza di **3** segmenti di dati chiamati **DSEG1**, **DSEG2** e **DSEG3**; subito dopo questi tre segmenti, incontriamo la riga:
`DGROUP GROUP DSEG1, DSEG2, DSEG3`.
In questa riga, la direttiva **GROUP** raggruppa i tre segmenti **DSEG1**, **DSEG2** e **DSEG3**, e assegna al gruppo il nome **DGROUP**; affinche' sia possibile fare una cosa del genere, e' necessario che la somma delle dimensioni dei tre segmenti sia inferiore a **65536** byte. A questo punto e' possibile utilizzare **DGROUP** per referenziare i tre segmenti di dati; a tale proposito sono presenti le istruzioni:

```
mov      ax, DGROUP     ; trasferisce DGROUP
mov      es: ax         ; in es
assume   es: DGROUP     ; assegna DGROUP a es
```

In questo modo, tutti gli indirizzamenti che coinvolgono dati definiti in **DSEG1**, **DSEG2** o **DSEG3** vengono calcolati rispetto a **ES=DGROUP**; come si puo' notare, sono presenti anche le istruzioni:

```
mov      ax, DATASEGM   ; trasferisce DATASEGM
mov      ds: ax         ; in ds
assume   ds: DATASEGM   ; assegna DATASEGM a ds
```

Queste istruzioni sono indispensabili per il corretto funzionamento della libreria **EXELIB**; infatti, le varie procedure definite in **EXELIB** si aspettano appunto **DS=DATASEGM**.
Analizziamo ora la struttura che assume il blocco **DGROUP**; prima di tutto osserviamo che **DSEG1**, **DSEG2** e **DSEG3** hanno tutti l'attributo di allineamento **PARA**, e quindi vengono sistemati in memoria a partire da indirizzi fisici multipli di **16**. Supponiamo che **DSEG1** venga caricato in memoria all'indirizzo fisico **01000h**; in questo caso a **DSEG1** viene assegnato il paragrafo **0100h**, e quindi **varWord1** si viene a trovare all'offset **0002h** rispetto a **DSEG1**. Successivamente **DSEG2** viene caricato in memoria all'indirizzo fisico **01010h** (cioe' a **01000h + 10h**); in questo caso a **DSEG2** viene assegnato il paragrafo **0101h**, e quindi **varWord2** si viene a

trovare all'offset **0004h** rispetto a **DSEG2**. Infine **DSEG3** viene caricato in memoria all'indirizzo fisico **01020h** (cioe' a **01010h + 10h**); in questo caso a **DSEG3** viene assegnato il paragrafo **0102h**, e quindi **varWord3** si viene a trovare all'offset **0006h** rispetto a **DSEG3**.
Il blocco **DGROUP** rappresenta allora questi tre segmenti disposti in memoria nel modo che abbiamo appena visto; l'inizio di **DGROUP** coincide ovviamente con l'inizio di **DSEG1**, mentre la fine di **DGROUP** coincide ovviamente con la fine di **DSEG3**. In sostanza si ottiene **DGROUP = DSEG1 = 0100h**; di conseguenza, rispetto a **DGROUP** la variabile **varWord1** si trova all'offset **0002h**, la variabile **varWord2** si trova all'offset **0014h**, la variabile **varWord3** si trova all'offset **0026h**.
Per dimostrare le cose appena dette, nel programma di Figura 17 notiamo la presenza dei primi tre gruppi di istruzioni che usano l'istruzione **LEA** per ottenere gli offset di **varWord1**, **varWord2** e **varWord3**, che vengono automaticamente calcolati rispetto a **DGROUP**; questi offset vengono poi visualizzati sullo schermo attraverso la procedura **writeHex16**.
I successivi tre gruppi di istruzioni fanno la stessa cosa utilizzando pero' la direttiva **OFFSET**; lo scopo di questi tre gruppi di istruzioni e' quello di illustrare un famoso bug che caratterizzava le vecchie versioni del **MASM**. In pratica, a causa di questo bug, la direttiva **OFFSET** calcola gli offset di **varWord1**, **varWord2** e **varWord3** rispetto ai segmenti di appartenenza **DSEG1**, **DSEG2** e **DSEG3**, e non rispetto a **DGROUP**; se si sta utilizzando una vecchia versione del **MASM**, si nota che il programma di Figura 17 visualizza per le tre variabili gli offset **0002h**, **0004h** e **0006h**. Il **Borland TASM** dispone di una modalita' operativa chiamata **Ideal**; in questa modalita' il **TASM** elimina il bug del **MASM**. Nella modalita' ordinaria il **TASM** emula il **MASM**, e i programmatori della **Borland** hanno deciso di emulare anche il bug del **MASM**; si tratta di una scelta abbastanza discutibile visto che ormai questo problema e' stato eliminato gia' da parecchio tempo (**MASM 6.0** del **1992**).
Il programma di Figura 17 illustra anche il metodo che si segue per evitare il bug del **MASM**; in pratica, o si utilizza **LEA**, oppure si utilizza la direttiva **OFFSET** con un segment override che specifica il segmento di programma rispetto al quale vogliamo calcolare l'offset. In Figura 17 vediamo appunto che viene applicato anche questo metodo che consiste nel ricorrere ad istruzioni del tipo:

```
mov ax, offset DGROUP:varWord1.
```

Attraverso il **TASM** o le vecchissime versioni del **MASM** e' possibile vedere che grazie a questo espediente vengono visualizzati gli offset corretti, calcolati rispetto a **DGROUP**.

E' importante ribadire che la direttiva **GROUP** raggruppa i vari segmenti di programma tenendo conto del modo con cui il programmatore ha stabilito la disposizione in memoria dei segmenti stessi; per chiarire questo aspetto analizziamo il programma di Figura 18.

Figura 18 - Uso della direttiva GROUP

```
;--------------------------------------------------;
; File group2.asm                                  ;
; Programma Assembly con la direttiva GROUP        ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                            ; set di istruzioni a 32 bit
INCLUDE     EXELIB.INC          ; inclusione libreria di I/O

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h              ; 1024 byte per lo stack

;############### segmento dati 1 ##################
```

```
DSEG1       SEGMENT  PARA PUBLIC USE16 'DATA'

varWord1    dw    31800

DSEG1       ENDS

;############### segmento dati 2 ##################

DSEG2       SEGMENT  PARA PUBLIC USE16 'DATA'

varWord2    dw    12900

DSEG2       ENDS

;############### segmento dati 3 ##################

DSEG3       SEGMENT  PARA PUBLIC USE16 'DATA'

varWord3    dw    11600

DSEG3       ENDS

;############### segmento dati 4 ##################

DSEG4       SEGMENT  PARA PUBLIC USE16 'DATA'

varWord4    dw    22950

DSEG4       ENDS

;############### segmento dati 5 ##################

DSEG5       SEGMENT  PARA PUBLIC USE16 'DATA'

varWord5    dw    10300

DSEG5       ENDS

;################ gruppo dgroup ###################

DGROUP      GROUP    DSEG1, DSEG3, DSEG5

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

   mov      ax, DGROUP              ; trasferisce DGROUP
   mov      es, ax                  ; in ES attraverso AX
   assume   es: DGROUP              ; assegna DGROUP a ES

;------- inizio blocco principale istruzioni ------

   call     clearScreen             ; pulizia schermo
```

```
   mov      dx, 0400h              ; riga 04h, colonna 00h
   lea      ax, varWord1           ; ax = offset varWord1
   call     writeHex16             ; visualizza ax

   inc      dh                     ; incremento riga
   lea      ax, varWord2           ; ax = offset varWord2
   call     writeHex16             ; visualizza ax

   inc      dh                     ; incremento riga
   lea      ax, varWord3           ; ax = offset varWord3
   call     writeHex16             ; visualizza ax

   inc      dh                     ; incremento riga
   lea      ax, varWord4           ; ax = offset varWord4
   call     writeHex16             ; visualizza ax

   inc      dh                     ; incremento riga
   lea      ax, varWord5           ; ax = offset varWord5
   call     writeHex16             ; visualizza ax

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                ; codice di uscita
   mov      ah, 4ch                ; servizio Return to DOS
   int      21h                    ; chiama i servizi DOS

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

In questo programma sono presenti **5** segmenti di dati chiamati **DSEG1**, **DSEG2**, **DSEG3**, **DSEG4** e **DSEG5**; tre di questi segmenti, e cioe' **DSEG1**, **DSEG3** e **DSEG5** vengono inseriti in un gruppo chiamato **DGROUP**. Per capire quale struttura viene assunta da **DGROUP**, cominciamo con l'osservare che tutti i **5** segmenti di dati hanno l'attributo di allineamento di tipo **PARA**; supponendo che il **SO** faccia partire **DSEG1** dall'indirizzo fisico **01000h**, allora **DSEG2** parte da **01010h**, **DSEG3** parte da **01020h**, **DSEG4** parte da **01030h** e **DSEG5** parte da **01040h**. In termini di paragrafi possiamo dire quindi che **DSEG1** parte dal paragrafo **0100h**, **DSEG2** parte dal paragrafo **0101h**, **DSEG3** parte dal paragrafo **0102h**, **DSEG4** parte dal paragrafo **0103h** e **DSEG5** parte dal paragrafo **0105h**; i segmenti **DSEG1**, **DSEG3** e **DSEG5** vengono sistemati in **DGROUP** con i requisiti di allineamento appena illustrati. In pratica, l'inizio di **DGROUP** coincide con l'inizio di **DSEG1**, e quindi **DGROUP = DSEG1 = 0100h**; abbiamo inoltre **DSEG3=0102h** e **DSEG5=0104h** con la fine di **DGROUP** che coincide con la fine di **DSEG5**. Il programma di Figura 18 visualizza gli offset delle **5** variabili definite nei **5** segmenti di dati del programma; eseguendo il programma si nota che gli offset di **varWord2** e **varWord4** vengono calcolati rispetto ai segmenti di appartenenza (**DSEG2** e **DSEG4**), mentre gli offset di **varWord1 varWord3** e **varWord5** vengono calcolati rispetto a **DGROUP**. Anche senza eseguire il programma, si intuisce subito che:

***** l'offset di **varWord1** vale **0000h** rispetto a **DSEG1** e **0000h** rispetto a **DGROUP**;

***** l'offset di **varWord3** vale **0000h** rispetto a **DSEG3** e **0020h** rispetto a **DGROUP**;
***** l'offset di **varWord5** vale **0000h** rispetto a **DSEG5** e **0040h** rispetto a **DGROUP**.

Un'ultima considerazione riguarda il fatto che chiaramente la direttiva **GROUP** dovrebbe essere utilizzata per raggruppare segmenti di programma dello stesso tipo; non avrebbe molto senso raggruppare ad esempio segmenti di dati con segmenti di codice. I compilatori **C/C++** usano spesso la direttiva **GROUP** per raggruppare dati statici e dati temporanei (stack); infatti, nella gran parte dei casi il blocco stack e il blocco dati principale di un programma non raggiungono assieme i **65536** byte di memoria necessaria. Se il programmatore vuole abilitare questa possibilita', deve attivare una apposita opzione dal menu di configurazione del compilatore; nel caso del **Borland C/C++ 3.1** questa opzione viene attivata selezionando **Assume SS Equals DS** dal menu **Options**. Raggruppando lo stack e il blocco dati principale, i compilatori **C/C++** riescono spesso a gestire in modo piu' efficiente un programma; se vogliamo fare una cosa del genere in Assembly possiamo basarci sull'esempio di Figura 19.

Figura 19 - Uso della direttiva GROUP

```
;--------------------------------------------------;
; File group3.asm                                  ;
; Programma Assembly con la direttiva GROUP        ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                  ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA

; definizione dati statici del programma

DATASEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

vstack      db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;################ gruppo dgroup ###################

DGROUP      GROUP    DATASEGM, STACKSEGM

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

start:                              ; entry point

   cli                              ; disabilita le INT hardware
```

```
   mov      ax, DGROUP              ; trasferisce DGROUP
   mov      ds, ax                  ; in DS
   mov      ss, ax                  ; e in SS
   assume   ds: DGROUP              ; assegna DGROUP a DS
   assume   ss: DGROUP              ; assegna DGROUP a SS
   mov      sp, offset DGROUP:vstack + STACKSIZE
   sti                              ; riabilita le INT hardware

;------- inizio blocco principale istruzioni ------

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; servizio Return to DOS
   int      21h                     ; chiama i servizi DOS

CODESEGM    ENDS

;###################################################

   END      start
```

Al momento di caricare il programma in memoria, il **SO** pone **CS=CODESEGM** e fa puntare **IP** all'entry point **start**; inoltre il **SO** pone **SS=STACKSEGM** e fa puntare **SP** alla fine di **STACKSEGM**, cioe' all'offset **0400h** calcolato rispetto a **STACKSEGM**.
Appena il programmatore riceve il controllo pone **DS=SS=DGROUP**; in questo modo i dati statici e i dati temporanei del programma possono essere gestiti indifferentemente con indirizzamenti calcolati rispetto a **DS** o a **SS**. Anche **SP** deve essere modificato in modo che punti all'offset successivo alla fine di **vstack**; quest'offset deve essere calcolato rispetto a **DGROUP** e non rispetto a **STACKSEGM** (come fa il **SO**).
Naturalmente e' fondamentale che **DATASEGM** e **STACKSEGM** vengano disposti come si vede in Figura 19; in questo modo i dati statici si trovano all'inizio di **DGROUP**, mentre i dati temporanei vengono disposti a partire dalla fine di **DGROUP**.
Nota importante.
Se si sta utilizzando la libreria **EXELIB**, si sconsigliano vivamente i raggruppamenti di qualsiasi genere; infatti **EXELIB** presuppone di essere linkata ad un programma dotato almeno di tre segmenti principali chiamati **DATASEGM**, **CODESEGM** e **STACKSEGM**, disposti in questo stesso ordine.

Capitolo 28 - Linking tra moduli Assembly

I numerosi programmi Assembly presentati nei precedenti capitoli, hanno in comune il fatto che il codice sorgente di ciascuno di essi e' interamente contenuto in un unico file; in questo capitolo viene illustrato il procedimento che bisogna seguire per poter distribuire su due o piu' files il codice sorgente di un programma Assembly. Ciascuno dei files che formano un programma viene definito **modulo**; il problema fondamentale che dobbiamo affrontare consiste nello stabilire i criteri in base ai quali i vari moduli comunicano tra loro.
La possibilita' di ripartire un programma su due o piu' files e' molto importante in quanto ci permette ad esempio di creare utili librerie di procedure che possiamo poi linkare a tutti i nostri programmi che ne hanno bisogno; in questo modo si risparmia parecchio lavoro in quanto si evita di riscrivere ogni volta tutte le procedure gia' presenti all'interno di una libreria. Un'altro esempio importante e' rappresentato dalla eventualita' di dover interfacciare l'Assembly con i linguaggi di alto livello; anche in questo caso infatti, dobbiamo essere in grado di far dialogare tra loro moduli contenenti codice Assembly con moduli contenenti codice **C**, **Pascal**, **BASIC**, etc.

Programmi Assembly contenuti in un unico file.

Per una migliore comprensione dei concetti esposti in questo capitolo, riassumiamo alcuni importanti aspetti legati alle fasi di **assembling** e di **linking** di un programma Assembly contenuto in un unico file; a tale proposito ci serviamo del programma di esempio **TESTMAIN.ASM** presentato in Figura 1.

Figura 1 - Esempio TESTMAIN.ASM

```
;--------------------------------------------------;
; File testmain.asm                                ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                  ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa1    db    'Stringa1 definita in DATASEGM', 10, 13, '$'
stringa2    db    'Stringa2 definita in DATASEGM', 10, 13, '$'
stringa3    db    'Stringa3 definita in DATASEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM    ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS
```

```
start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   push     offset stringa1         ; indirizzo stringa1
   call     near ptr printStr       ; visualizza stringa1

   push     offset stringa2         ; indirizzo stringa2
   call     near ptr printStr       ; visualizza stringa2

   push     offset stringa3         ; indirizzo stringa3
   call     near ptr printStr       ; visualizza stringa3

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; servizio Terminate Process
   int      21h                     ; chiama i servizi DOS

;------------- inizio blocco procedure ------------

; procedure printStr(var strAddr: String)
; convenzioni Pascal

printStr proc near

   strAddr  equu   [bp+4]           ; parametro strAddr

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:bp

   mov      dx, strAddr             ; ds:dx punta a strAddr
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      bp                      ; ripristina bp
   ret      2                       ; return + pulizia stack

printStr endp

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

Nel blocco codice **CODESEGM** di questo programma abbiamo una procedura **printStr** che utilizza il servizio **09h** dell'**INT 21h** per visualizzare una stringa; la procedura **printStr** segue le convenzioni **Pascal**, e quindi ha la responsabilita' di ripulire lo stack dagli argomenti ricevuti. Nel blocco dati **DATASEGM** vengono definite tre stringhe **DOS**; nel blocco codice principale queste tre stringhe vengono passate a **printStr** per essere visualizzate. La procedura **printStr** richiede il solo offset (indirizzo **NEAR**) della stringa da visualizzare, e presuppone quindi che **DS** contenga gia' il segmento di programma a cui appartiene la stringa stessa; proprio per questo motivo e' importantissimo che il programmatore, prima di chiamare **printStr**, inizializzi opportunamente **DS**. Siccome le tre stringhe vengono definite in **DATASEGM**, l'inizializzazione consiste nel porre **DS=DATASEGM**; in Figura 1 la direttiva **ASSUME** che associa **DS** a **DATASEGM** e' superflua in quanto stiamo accedendo per indirizzo e non per nome ai dati del programma. Bisogna ricordare infatti che gli operatori **SEG** e **OFFSET** fanno riferimento direttamente al segmento di apparteneza del loro operando e non ai registri di segmento.

Una volta che abbiamo scritto il codice sorgente, possiamo passarlo all'assembler; la prima fase svolta dall'assembler consiste nel convertire in codice macchina il contenuto di ciascun segmento di programma. In questa fase l'assembler determina l'offset e la dimensione in byte di ogni singolo dato e di ogni singola istruzione; prima di tutto l'assembler codifica il blocco **DATASEGM** ottenendo la situazione mostrata in Figura 2.

Figura 2 - Assemblaggio di DATASEGM

```
0000h    DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

0000h    53 74 72 69 6E 67 61 31 20 64 65 66 69 6E 69 74
         61 20 69 6E 20 44 41 54 41 53 45 47 4D 0A 0D 24
0020h    53 74 72 69 6E 67 61 32 20 64 65 66 69 6E 69 74
         61 20 69 6E 20 44 41 54 41 53 45 47 4D 0A 0D 24
0040h    53 74 72 69 6E 67 61 33 20 64 65 66 69 6E 69 74
         61 20 69 6E 20 44 41 54 41 53 45 47 4D 0A 0D 24

0060h    DATASEGM    ENDS
```

Convenzionalmente l'assembler fa partire dal paragrafo **0000h** tutti i segmenti di programma da assemblare; il segmento **DATASEGM** e' allineato al paragrafo per cui il location counter **$** inizia a contare dall'offset **0000h**. L'assembler rileva che il primo dato (**stringa1**) si trova appunto all'offset **0000h** ed e' un vettore di **20h** byte che occupano tutti gli offset compresi tra **0000h** e **001Fh**; come si puo' notare, trattandosi di una stringa l'assembler converte tutti i caratteri nei corrispondenti codici ASCII. Successivamente l'assembler incontra il dato **stringa2** e rileva che si tratta di un vettore di **20h** byte che parte dall'offset **0020h**; infine questo stesso lavoro viene ripetuto per **stringa3** che parte dall'offset **0040h** ed e' formata da **20h** byte. L'aspetto importante da ribadire e' dato dal fatto che dopo l'assemblaggio, ogni dato viene convertito in tre parametri fondamentali che sono: l'offset di partenza, il numero di byte occupati in memoria e il contenuto binario di ogni singolo byte; siccome il blocco **DATASEGM** e' interamente contenuto nel file **TESTMAIN.ASM**, l'assembler e' in grado di individuare offset, dimensione e contenuto di ogni singolo dato appartenente a questo blocco. Complessivamente il blocco **DATASEGM** occupa in memoria **60h** byte.

Terminato l'assemblaggio del blocco **DATASEGM** l'assembler incontra il blocco **CODESEGM**; l'assemblaggio del blocco **CODESEGM** produce il risultato mostrato in Figura 3.

Figura 3 - Assemblaggio di CODESEGM

```
0000h    CODESEGM    SEGMENT  PARA PUBLIC USE16      'CODE'

0000h    B8 0000s       ; mov    ax, DATASEGM
0003h    8E D8          ; mov    ds, ax

0005h    68 0000r       ; push   offset stringa1
0008h    E8 0012        ; call   near ptr printStr

000Bh    68 0020r       ; push   offset stringa2
000Eh    E8 000C        ; call   near ptr printStr

0011h    68 0040r       ; push   offset stringa3
0014h    E8 0006        ; call   near ptr printStr

0017h    B0 00          ; mov    al, 00h
0019h    B4 4C          ; mov    ah, 4ch
001Bh    CD 21          ; int    21h

001Dh    55             ; push   bp
001Eh    8B EC          ; mov    bp, sp
0020h    8B 56 04       ; mov    dx, strAddr
0023h    B4 09          ; mov    ah, 09h
0025h    CD 21          ; int    21h
0027h    5D             ; pop    bp
0028h    C2 0002        ; ret    2

002Bh    CODESEGM    ENDS
```

Anche in questo caso possiamo notare come l'assembler faccia partire idealmente il blocco **CODESEGM** dal paragrafo **0000h** della **RAM**; siccome **CODESEGM** e' allineato al paragrafo, il location counter **$** inizia a contare dall'offset **0000h**. Nella fase di assemblaggio di **CODESEGM**, l'assembler determina l'offset e il codice macchina di ciascuna istruzione; ogni riferimento a identificatori di variabili, procedure, etichette, etc, viene sostituito con il corrispondente indirizzo di memoria. Naturalmente cio' e' possibile solo se l'assembler conosce gia' l'indirizzo dell'identificatore; questa situazione si verifica quando l'identificatore viene definito nello stesso file che l'assembler sta assemblando. Consideriamo ad esempio in Figura 3 l'istruzione:

`push offset stringa2`;

in questo caso l'assembler e' in grado di determinare che l'offse di **stringa2** e' **0020h**, e produce quindi il codice macchina:

`68 0020r`.

La **r** sta per **relocatable** (rilocabile) ed e' una informazione che l'assembler passa al linker; questa informazione e' necessaria quando il blocco **DATASEGM** si trova distribuito su due o piu' files. In un caso del genere il linker deve fondere tra loro i vari blocchi **DATASEGM** provvedendo poi a modificare (rilocare) gli offset delle informazioni contenute nel blocco risultante; nel caso del nostro esempio tutto il programma e' contenuto all'interno di un unico file, per cui abbiamo la certezza che l'offset **0020h** di **stringa1** non verra' rilocato (tutti questi aspetti vengono descritti piu' avanti).

Consideriamo ora in Figura 3 l'istruzione:

`mov ax, DATASEGM`;

per poter convertire questa istruzione in codice macchina l'assembler assegna a **DATASEGM** il paragrafo simbolico **0000s**. La **s** sta per **segment** e indica il fatto che si tratta appunto di un

paragrafo simbolico (il **MASM** utilizza la **r** anche per i segmenti rilocabili); il vero paragrafo verra' assegnato a **DATASEGM** dal **SO** al momento di caricare il programma in memoria. Complessivamente il blocco **CODESEGM** richiede **2Bh** byte di memoria.

L'ultimo segmento di programma incontrato dall'assembler e' **STACKSEGM**; l'assemblaggio di **STACKSEGM** produce il risultato mostrato in Figura 4.

Figura 4 - Assemblaggio di STACKSEGM

```
0000h    STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

0000h    0400*(??)

0400h    STACKSEGM   ENDS
```

L'assembler rileva che nel blocco **STACKSEGM** viene definito un vettore di **400h** byte non inizializzati, che occupano tutti gli offset compresi tra **0000h** e **03FFh**; la dimensione complessiva di **STACKSEGM** ammonta quindi a **400h** byte (cioe' **3FFh** byte piu' il byte che si trova a **0000h**).

In seguito alla fase di assemblaggio l'assembler raccoglie tutte le informazioni relative agli identificatori presenti nel programma; queste informazioni vengono inserite dall'assembler nella **Symbol Table** (tabella dei simboli) illustrata in Figura 5.

Figura 5 - Symbol Table di TESTMAIN.ASM

```
Symbol Name        Type         Value

STACK_SIZE         Number       0400h
printStr           Near         CODESEGM:001Dh
start              Near         CODESEGM:0000h
strAddr            Text         [bp+4]
stringa1           Byte         DATASEGM:0000h
stringa2           Byte         DATASEGM:0020h
stringa3           Byte         DATASEGM:0040h

Groups & Segments     Bit  Size    Align   Combine Class

CODESEGM              16   002Bh   Para    Public  CODE
DATASEGM              16   0060h   Para    Public  DATA
STACKSEGM             16   0400h   Para    Stack   STACK
```

Ad ogni identificatore (**symbol**) l'assembler associa il tipo e il valore; per le costanti come **STACK_SIZE** il tipo e' **Number**. Per i dati come **stringa1** il tipo si riferisce ad ogni singolo elemento (**Byte**, **Word**, **Dword**, etc), mentre il valore indica l'indirizzo logico **seg:offset** del dato stesso; per le etichette (compresi i nomi di procedure) il tipo si riferisce alla modalita' di indirizzamento (**Near** o **Far**), mentre il valore indica l'indirizzo logico **seg:offset** dell'etichetta stessa. Per le macro alfanumeriche come **strAddr** viene usato il tipo **Text**; il valore di una macro alfanumerica non e' altro che il corpo della macro stessa.
La **Symbol Table** termina con una serie di dettagliate informazioni relative ai vari segmenti di programma; nel caso di Figura 5 possiamo constatare che grazie al fatto che **TESTMAIN.ASM** e' contenuto in un unico file, l'assembler ha potuto determinare in modo completo tutte le caratteristiche di ogni singolo identificatore. A questo punto l'assembler genera il file **TESTMAIN.OBJ** (object file) contenente il codice macchina di **TESTMAIN.ASM** e la relativa **Symbol Table**; tutte le informazioni contenute in **TESTMAIN.OBJ** possono essere ora passate al linker.

Nello svolgere il proprio lavoro il linker segue un comportamento ben definito; se impartiamo ad esempio il comando:

```
link FILE1.OBJ + FILE2.OBJ + FILE3.OBJ,
```

questi tre files vengono esaminati dal linker nello stesso ordine da noi stabilito. In assenza di diverse indicazioni da parte del programmatore, il linker rispetta la disposizione dei segmenti da noi stabilita (disposizione sequenziale); piu' avanti vengono illustrati gli strumenti che l'assembler mette a disposizione per stabilire i criteri di ordinamento dei segmenti di programma. Nel caso del nostro esempio il linker comincia ad esaminare il primo segmento di programma del primo file **FILE1.OBJ**; il linker verifica poi se in **FILE1.OBJ**, **FILE2.OBJ** o **FILE3.OBJ** ci sono altri segmenti di programma aventi lo stesso nome, lo stesso attributo di combinazione (diverso da **PRIVATE**) e lo stesso attributo di classe del segmento che sta esaminando. In caso affermativo tutti questi segmenti vengono fusi in un unico segmento rispettando gli attributi di allineamento dei singoli segmenti (la dimensione del segmento finale non deve superare i **65536** byte pari a **64** Kb); in caso di fusione tra due o piu' segmenti, tutti i segmenti successivi al primo subiscono la rilocazione degli offset (come viene mostrato piu' avanti).
A questo punto il linker cerca nei tre object files altri segmenti aventi lo stesso attributo di classe di quello appena esaminato; in pratica, se il linker incontra per primo un segmento di classe **'DATA'**, va a cercare tutti gli altri segmenti di classe **'DATA'** e li dispone in modo consecutivo e contiguo nell'eseguibile finale. Se il linker incontra invece per primo un segmento di classe **'CODE'**, va a cercare tutti gli altri segmenti di classe **'CODE'** e li dispone in modo consecutivo e contiguo nell'eseguibile finale; tutte queste fasi vengono svolte su tutti i segmenti di programma presenti in tutti gli object files passati al linker. Il nome assegnato all'eseguibile finale e' quello del primo modulo esaminato dal linker; nol nostro caso otteniamo un eseguibile chiamato **FILE1.EXE**.

Nel caso dell'esempio di Figura 1, esiste un unico object file che e' **TESTMAIN.OBJ**; in un caso del genere il lavoro svolto dal linker viene notevolmente semplificato. Il linker inizia ad esaminare per primo il blocco **DATASEGM** che ha combinazione **PUBLIC** e classe **'DATA'**; non esistendo altri blocchi con queste tre caratteristiche, il segmento **DATASEGM** non viene fuso con altri segmenti e rimane quindi cosi' com'e'.
Il linker cerca poi altri blocchi con classe **'DATA'**; non esistendo altri blocchi del genere, il linker passa direttamente al segmento **CODESEGM** che ha combinazione **PUBLIC** e classe **'CODE'**. Anche il blocco **CODESEGM** non subisce nessuna fusione, e non viene neanche raggruppato con altri blocchi di classe **'CODE'**; lo stesso discorso vale anche per l'ultimo segmento esaminato dal linker, e cioe' per **STACKSEGM**.
I tre segmenti di programma di Figura 1 vengono disposti nell'eseguibile finale secondo l'ordine imposto dal programmatore, e nel rispetto dei singoli attributi di allineamento; il lavoro complessivo svolto dal linker viene rappresentato simbolicamente dal **Map File** di Figura 6.

Figura 6 - Map File TESTMAIN.MAP

```
Start    Stop     Length   Name        Class

00000h   0005Fh   00060h   DATASEGM    DATA
00060h   0008Ah   0002Bh   CODESEGM    CODE
00090h   0048Fh   00400h   STACKSEGM   STACK

Program entry point at 0006h:0000h
```

Il primo segmento di programma incontrato dal linker e' **DATASEGM**; come indirizzo fisico di partenza viene utilizzato simbolicamente **00000h**. Siccome **DATASEGM** ha l'attributo di allineamento **PARA**, il linker fa partire questo blocco dall'indirizzo logico **0000h:0000h**; di conseguenza si ottiene **DATASEGM=0000h**, mentre l'offset iniziale e' **0000h**. Come si puo' notare, la lunghezza complessiva di **DATASEGM** e' di **60h** byte, pari alla lunghezza gia' calcolata

dall'assembler; questo significa che il blocco **DATASEGM** di **TESTMAIN.OBJ** non e' stato fuso con nessun altro blocco, e quindi non c'e' stata da parte del linker nessuna rilocazione degli offset. In base a queste considerazioni possiamo dire che **DATASEGM** parte dall'indirizzo fisico **00000h** e termina all'indirizzo fisico **0005Fh**.
Il secondo segmento di programma incontrato dal linker e' **CODESEGM** per il quale abbiamo richiesto l'allineamento al paragrafo; l'indirizzo fisico multiplo di **16** immediatamente successivo a **0005Fh** e' **00060h**. Il linker fa partire quindi **CODESEGM** dall'indirizzo logico **0006h:0000h**; di conseguenza si ottiene **CODESEGM=0006h**, mentre l'offset iniziale e' **0000h**. Come si puo' notare, la lunghezza complessiva di **CODESEGM** e' di **2Bh** byte, pari alla lunghezza gia' calcolata dall'assembler; questo significa che il blocco **CODESEGM** di **TESTMAIN.OBJ** non e' stato fuso con nessun altro blocco, e quindi non c'e' stata da parte del linker nessuna rilocazione degli offset. In definitiva possiamo dire che **CODESEGM** parte dall'indirizzo fisico **00060h** e termina all'indirizzo fisico **0008Ah**.
Il terzo segmento di programma incontrato dal linker e' **STACKSEGM** per il quale abbiamo richiesto l'allineamento al paragrafo; l'indirizzo fisico multiplo di **16** immediatamente successivo a **0008Ah** e' **00090h**. Il linker fa partire quindi **STACKSEGM** dall'indirizzo logico **0009h:0000h**; di conseguenza si ottiene **STACKSEGM=0009h**, mentre l'offset iniziale e' **0000h**. Come si puo' notare, la lunghezza complessiva di **STACKSEGM** e' di **400h** byte, pari alla lunghezza gia' calcolata dall'assembler; questo significa che il blocco **STACKSEGM** di **TESTMAIN.OBJ** non e' stato fuso con nessun altro blocco, e quindi non c'e' stata da parte del linker nessuna rilocazione degli offset. In definitiva possiamo dire che **STACKSEGM** parte dall'indirizzo fisico **00090h** e termina all'indirizzo fisico **0048Fh**.

Naturalmente il linker ricava dalla **Symbol Table** tutte le informazioni relative ai vari segmenti di programma e agli indentificatori; in questo caso particolare, l'intero programma e' contenuto nel file **TESTMAIN.OBJ**, per cui il linker ha dovuto svolgere un lavoro relativamente semplice. Il passo finale compiuto dal linker consiste nella generazione del file eseguibile **TESTMAIN.EXE**; come e' stato spiegato nel Capitolo 13, un file eseguibile in formato **EXE** e' formato da una intestazione (**header**) seguita dal programma vero e proprio. All'interno dell'intestazione sono presenti importanti informazioni destinate al **SO** come ad esempio le informazioni di rilocazione relative ai vari segmenti di programma; in questo modo il linker dice al **SO** che **DATASEGM** parte da **0000h:0000h**, **CODESEGM** parte da **0006h:0000h** e **STACKSEGM** parte da **0009h:0000h**. Ossevando la Figura 3 si puo' notare che l'entry point **start** si trova all'indirizzo logico **CODESEGM:0000h = 0006h:0000h**; di conseguenza il linker pone nell'**header**: **InitialCS=0006h** e **InitialIP=0000h** (Figura 6). Il linker nota anche la presenza di un segmento di programma **STACKSEGM=0009h** con attributo di combinazione **STACK** e con lunghezza pari a **400h** byte; di conseguenza il linker pone nell'**header**: **InitialSS=0009h** e **InitialSP=0400h**.

Il compito di caricare **TESTMAIN.EXE** in memoria spetta al **loader** (caricatore) del **SO**; prima di tutto il **loader** richiede due blocchi di memoria che come sappiamo vengono chiamati **Environment Segment** e **Program Segment**. In particolare nel **Program Segment** vengono inseriti i **256** byte del **PSP**, immediatamente seguiti dal programma vero e proprio; nel caso di **TESTMAIN.EXE** il programma e' formato dai tre blocchi visibili in Figura 6. I blocchi di memoria forniti dal **DOS** sono sempre allineati al paragrafo; supponiamo allora che il **DOS** fornisca al **loader** un **Program Segment** che parte dall'indirizzo fisico **0AC20h**, e cioe' dall'indirizzo logico **0AC2h:0000h**. Il **loader** procede a questo punto alla rilocazione dei segmenti di programma di **TESTMAIN.EXE**; i **256** byte (**100h**) del **PSP** occupano tutti gli indirizzi fisici compresi tra **0AC20h** e **0AD1Fh**, per cui il programma vero e proprio partira' dall'indirizzo fisico **0AD20h**, e cioe' dal paragrafo **0AD2h**. I registri **DS** e **ES** vengono inizializzati con il paragrafo di memoria assegnato al **PSP**, per cui si ottiene:

```
DS = ES = PSP = 0AC2h.
```

Il registro **CS** viene inizializzato con il valore rilocato di **InitialCS**; si ottiene quindi (Figura 6):
`CS = InitialCS + 0AD2h = 0006h + 0AD2h = 0AD8h`.
Il registro **IP** viene inizializzato con il valore di **InitialIP**; si ottiene quindi:
`IP = InitialIP = 0000h`.
Il registro **SS** viene inizializzato con il valore rilocato di **InitialSS**; si ottiene quindi (Figura 6):
`SS = InitialSS + 0AD2h = 0009h + 0AD2h = 0ADBh`.
Il registro **SP** viene inizializzato con il valore di **InitialSP**; si ottiene quindi:
`SP = InitialSP = 0400h`.

Programmi Assembly distribuiti su piu' moduli.

Nell'esempio **TESTMAIN.ASM** e nei numerosi altri esempi presentati nei precedenti capitoli, abbiamo utilizzato molto spesso il servizio **Display String** dell'**INT 21h** che ci permette di visualizzare facilmente una stringa **DOS**; per evitare di dover riscrivere ogni volta il codice relativo al servizio **Display String**, possiamo pensare di utilizzare una apposita procedura (come **printStr**) inserita in un file esterno. Ogni volta che un nostro programma ha bisogno di questa procedura, non deve fare altro che chiamarla; il problema fondamentale che dobbiamo affrontare consiste nel dire all'assembler e al linker che questa procedura si trova in un file diverso da quello in cui si trova il caller.
Partiamo allora da **TESTMAIN.ASM** e spostiamo la procedura **printStr** in un blocco **CODESEGM** che si trova pero' in un file chiamato **TESTLIB1.ASM**; in questo file inseriamo anche una seconda procedura **setCursor** che ci permette di posizionare il cursore (**prompt**) sullo schermo. La procedura **setCursor** utilizza il servizio n. **02h** (**Set Cursor Position**) dell'**INT 10h** (**Video BIOS Services**); le caratteristiche di questo servizio sono le seguenti:

```
Servizio Set Cursor Position dell'INT 10h

AH = 02h (Set Cursor Position)
BH = pagina video
DH = riga (00h = prima riga)
DL = colonna (00h = prima colonna)
```

Per il momento e' sufficiente sapere che nel registro **BH** va inserito il valore **00h** che rappresenta la pagina video predefinita; informazioni dettagliate sulle interruzioni hardware e software vengono esposte nella sezione **Assembly Avanzato**.
Possiamo organizzare allora **setCursor** in modo che richieda due argomenti; questi due argomenti sono la riga e la colonna in cui vogliamo posizionare il cursore. La procedura **setCursor** puo' essere definita di tipo **NEAR** in quanto si trova nello stesso segmento **CODESEGM** in cui si trova anche il caller; in definitiva, la procedura **SetCursor** in versione **Pascal** assume la seguente struttura:

```
; procedure setCursor(riga, colonna: Integer)

setCursor proc near

   colonna      equ   [bp+4]       ; parametro colonna
   riga         equ   [bp+6]       ; parametro riga

   push         bp                 ; preserva bp
   mov          bp, sp             ; ss:bp = ss:sp

   mov          ah, 02h            ; ah = Set Cursor Position
   mov          bh, 00h            ; bh = pagina video
   mov          dh, riga           ; dh = riga
   mov          dl, colonna        ; dl = colonna
```

```
   int         10h              ; chiama i servizi Video BIOS

   pop         bp               ; ripristina bp
   ret         4                ; near return + pulizia stack

setCursor endp
```

La procedura **setCursor** segue le convenzioni **Pascal**, per cui i suoi parametri vengono inseriti nello stack a partire dal primo, e vengono quindi a trovarsi disposti in ordine inverso rispetto alla lista mostrata dal prototipo; questi due parametri occupano **2** byte ciascuno, e quindi **setCursor** prima di terminare deve togliere dallo stack **4** byte. I parametri **riga** e **colonna** sono entrambi a **16** bit in quanto non possiamo inserire valori a **8** bit nello stack; naturalmente, all'interno di **setCursor** vengono utilizzati solo gli **8** bit meno significativi di ciascun parametro.
Passiamo ora alla procedura **printStr**; per rendere **printStr** piu' sofisticata, facciamo in modo che questa procedura sia in grado di stampare una stringa in un punto preciso dello schermo. Per ottenere questo risultato **printStr** puo' chiamare proprio **setCursor**; la struttura di **printStr** in versione **Pascal** sara' allora la seguente:

```
; procedure printStr(riga, colonna: Integer; var strAddr: String)

printStr proc near

   strAddr     equ   [bp+4]     ; parametro strAddr
   colonna     equ   [bp+6]     ; parametro colonna
   riga        equ   [bp+8]     ; parametro riga

   push        bp               ; preserva bp
   mov         bp, sp           ; ss:bp = ss:sp

   push        word ptr riga     ; passa riga a setCursor
   push        word ptr colonna  ; passa colonna a setCursor
   call        near ptr setCursor; chiama setCursor

   mov         ah, 09h          ; ah = Display String
   mov         dx, strAddr      ; ds:dx punta a strAddr
   int         21h              ; chiama i servizi DOS

   pop         bp               ; ripristina bp
   ret         6                ; near return + pulizia stack

printStr endp
```

Anche **printStr** segue le convenzioni **Pascal**; per questa procedura valgono quindi tutte le considerazioni gia' svolte per **setCursor**. In particolare, e' importante notare la chiamata di **setCursor** effettuata da **printStr**; gli argomenti vengono passati a partire dal primo, mentre la pulizia dello stack viene delegata a **setCursor**.

A questo punto possiamo procedere con la separazione del programma **TESTMAIN.ASM** di Figura 1 in due moduli chiamati **TESTMAIN.ASM** e **TESTLIB1.ASM**; il file **TESTMAIN.ASM** contiene il blocco dati del programma, il blocco stack e l'entry point seguito dalle istruzioni principali. All'interno di **TESTLIB1.ASM** e' presente un'altro segmento **CODESEGM** contenente le due procedure **setCursor** e **printStr**; prima di passare alla scrittura del codice sorgente, e' necessario pero' esporre alcune importanti considerazioni relative agli identificatori di un programma.

In assenza di diverse indicazioni fornite dal programmatore, tutti gli identificatori definiti in un file hanno una visibilita' limitata al file stesso; questo significa che in un programma formato da piu' moduli, gli identificatori definiti in un modulo non possono essere visti dagli altri moduli. Di conseguenza e' possibile ridefinire uno stesso identificatore in moduli diversi senza che si verifichi nessuna interferenza; si tratta comunque di uno stile di programmazione da evitare in quanto puo' creare notevole confusione.
Consideriamo ora il programma **TESTMAIN.ASM** mostrato in Figura 1; nel blocco **DATASEGM** vengono definite staticamente tre stringhe alle quali vengono assegnati gli identificatori **stringa1**, **stringa2** e **stringa3**. A ciascuno di questi tre nomi il linker assegna un offset che rimane fisso (statico) per tutta la fase di esecuzione di **TESTMAIN.EXE**; inoltre, la visibilita' di ciascuno di questi nomi e' limitata al file **TESTMAIN.ASM**. Nel blocco **CODESEGM** vengono definite staticamente due etichette alle quali vengono assegnati gli identificatori **start** e **printStr**; anche questi due identificatori hanno una visibilita' limitata al file **TESTMAIN.ASM**, ed esistono per tutta la fase di esecuzione di **TESTMAIN.EXE**. Nel file **TESTMAIN.ASM** notiamo anche la presenza dell'identificatore **strAddr** che rappresenta il nome di una macro alfanumerica; anche **strAddr** ha una visibilita' limitata al solo file **TESTMAIN.ASM**. Tra l'identificatore di una macro alfanumerica e gli identificatori di variabili, etichette e procedure esiste pero' una differenza importante; infatti, come abbiamo visto nel Capitolo 23, una macro alfanumerica puo' essere ridefinita piu' volte anche all'interno dello stesso file. Il caso della macro numerica **STACK_SIZE** e' leggermente diverso; infatti, anche **STACK_SIZE** e' visibile solo all'interno di **TESTMAIN.ASM**, ma questo identificatore puo' essere ridefinito solo in un altro file e non in **TESTMAIN.ASM**.

Per chiarire meglio questi importanti concetti passiamo all'implementazione pratica del nostro programma formato dai due moduli **TESTMAIN.ASM** e **TESTLIB1.ASM**; all'interno del blocco **CODESEGM** di **TESTMAIN.ASM** e' possibile inserire le istruzioni per la chiamata di **setCursor** e **printStr** che si trovano in un altro blocco **CODESEGM** del modulo **TESTLIB1.ASM**. Per informare l'assembler che queste due procedure si trovano in un file esterno, dobbiamo utilizzare la direttiva **EXTRN**; il **MASM** e il **TASM 5.x** (**TASM32.EXE**) supportano anche l'uso della direttiva **EXTERN**. La direttiva **EXTRN** richiede una lista di argomenti separati da virgole; ogni argomento e' formato da un identificatore esterno, da un due punti (**:**) e dal tipo di identificatore. Il tipo di un identificatore e' quello specificato dall'assembler nella colonna **Type** della **Symbol Table**; in pratica, il tipo di una etichetta esterna o di una procedura esterna non e' altro che la modalita' di indirizzamento, e cioe' **NEAR** o **FAR**. Il tipo di una variabile esterna e' la dimensione di ogni suo singolo elemento e cioe' **Byte**, **Word**, **Dword**, etc; nel caso del modulo **TESTMAIN.ASM** dobbiamo scrivere quindi:

```
EXTRN setCursor: NEAR, printStr: NEAR.
```

Il punto del programma in cui deve essere inserita la direttiva **EXTRN** assume una importanza enorme; infatti l'assembler per poter generare il corretto codice macchina, ha bisogno di conoscere il tipo dell'identificatore esterno e il segmento di appartenenza. Per ottenere queste informazioni l'assembler fa riferimento al segmento di programma in cui viene inserita la direttiva **EXTRN**; e' importantissimo quindi che la direttiva **EXTRN** relativa ad un identificatore esterno, venga inserita nel segmento di programma in cui si trova la definizione dell'identificatore stesso. Nel caso del nostro esempio, la direttiva **EXTRN** per **setCursor** e per **printStr** deve essere inserita nel blocco **CODESEGM** in quanto questo e' il blocco in cui vengono definite le due procedure esterne; se non si seguono rigorosamente queste regole, si ottiene un programma che in genere manda in crash il computer. Per i motivi che vengono esposti piu' avanti, si consiglia di inserire le direttive **EXTRN** proprio all'inizio del segmento di programma di competenza; la Figura 7 illustra il nuovo aspetto che assume il modulo **TESTMAIN.ASM** del nostro esempio.

Figura 7 - Modulo TESTMAIN.ASM

```
;---------------------------------------------------;
; File testmain.asm                                 ;
;---------------------------------------------------;

;########### direttive per l'assembler ############

.386                               ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                 ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

;---------- inizio definizione variabili ----------

stringa1    db    'Stringa1 definita in DATASEGM', 10, 13, '$'
stringa2    db    'Stringa2 definita in DATASEGM', 10, 13, '$'
stringa3    db    'Stringa3 definita in DATASEGM', 10, 13, '$'

;----------- fine definizione variabili -----------

DATASEGM    ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   extrn    setCursor: near, printStr: near

   assume   cs: CODESEGM           ; assegna CODESEGM a CS


start:                             ; entry point

   mov      ax, DATASEGM           ; trasferisce DATASEGM
   mov      ds, ax                 ; in DS attraverso AX
   assume   ds: DATASEGM           ; assegna DATASEGM a DS

;------- inizio blocco principale istruzioni ------

   push     10                     ; riga 10
   push     0                      ; colonna 0
   push     offset stringa1        ; indirizzo stringa1
   call     near ptr printStr      ; visualizza stringa1

   push     12                     ; riga 11
   push     0                      ; colonna 0
   push     offset stringa2        ; indirizzo stringa2
   call     near ptr printStr      ; visualizza stringa2

   push     14                     ; riga 12
   push     0                      ; colonna 0
   push     offset stringa3        ; indirizzo stringa3
   call     near ptr printStr      ; visualizza stringa3

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                ; codice di uscita
```

```
   mov      ah, 4ch                  ; servizio Terminate Process
   int      21h                      ; chiama i servizi DOS

;------------- inizio blocco procedure ------------

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;###################################################

   END      start
```

Come si puo' notare, **printStr** viene chiamata in stile **Pascal**, per cui i tre argomenti vengono inseriti nello stack a partire dal primo, mentre la pulizia dello stack spetta alla stessa **printStr**; e' importante osservare anche il fatto che gli argomenti della direttiva **EXTRN** posta all'inizio di **CODESEGM**, possono essere visti come veri e propri prototipi delle procedure **setCursor** e **printStr**, e rendono quindi superfluo l'uso dell'operatore **NEAR PTR** nella chiamata delle procedure stesse.
Passiamo ora al modulo **TESTLIB1.ASM** contenente il blocco **CODESEGM** all'interno del quale vengono definite le due procedure **setCursor** e **printStr**; come e' stato detto in precedenza, in assenza di diverse indicazioni da parte del programmatore questi due identificatori vengono definiti staticamente in **CODESEGM** e risultano visibili solo all'interno del modulo **TESTLIB1.ASM**. Per fare in modo che i due identificatori **setCursor** e **printStr** risultino visibili anche all'esterno di **TESTLIB1.ASM**, dobbiamo renderli pubblici; a tale proposito l'assembler ci mette a disposizione la direttiva **PUBLIC**. Questa direttiva richiede una lista di argomenti separati da virgole; ciascun argomento e' costituito dall'identificatore che vogliamo rendere pubblico. Nel caso del modulo **TESTLIB1.ASM** dobbiamo scrivere quindi:
`PUBLIC setCursor, printStr.`
Il punto del programma in cui inserire la direttiva **PUBLIC** appare abbastanza ovvio; e' evidente infatti che la direttiva **PUBLIC** debba essere inserita nel segmento di programma che contiene le definizioni degli identificatori che vogliamo rendere pubblici. Anche in questo caso si consiglia di posizionare la direttiva **PUBLIC** proprio all'inizio del segmento di programma di competenza; in base alle considerazioni appena esposte, il modulo **TESTLIB1.ASM** assume l'aspetto mostrato in Figura 8.

Figura 8 - Modulo TESTLIB1.ASM

```
;--------------------------------------------------;
; File testlib1.asm                                ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                                 ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########
```

```
;############### segmento codice #################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   public   setCursor, printStr

   assume   cs: CODESEGM           ; assegna CODESEGM a CS

;------------- inizio blocco procedure ------------

; procedure setCursor(riga, colonna: Integer)

setCursor proc near

   colonna  equ   [bp+4]           ; parametro colonna
   riga     equ   [bp+6]           ; parametro riga

   push     bp                     ; preserva bp
   mov      bp, sp                 ; ss:bp = ss:bp

   mov      ah, 02h                ; ah = Set Cursor Position
   mov      bh, 00h                ; bh = pagina video 0
   mov      dh, riga               ; dh = riga
   mov      dl, colonna            ; dl = colonna
   int      10h                    ; chiama i servizi Video BIOS

   pop      bp                     ; ripristina bp
   ret      4                      ; return + pulizia stack

setCursor endp

; procedure printStr(riga, colonna: Integer; var strAddr: String)

printStr proc near

   strAddr  equ   [bp+4]           ; parametro strAddr
   colonna  equ   [bp+6]           ; parametro colonna
   riga     equ   [bp+8]           ; parametro riga

   push     bp                     ; preserva bp
   mov      bp, sp                 ; ss:bp = ss:bp

   push     word ptr riga          ; passa riga
   push     word ptr colonna       ; passa colonna
   call     near ptr setCursor     ; chiama setCursor

   mov      dx, strAddr            ; ds:dx punta a strAddr
   mov      ah, 09h                ; servizio Display String
   int      21h                    ; chiama i servizi DOS

   pop      bp                     ; ripristina bp
   ret      6                      ; return + pulizia stack

printStr endp

;-------------- fine blocco procedure -------------

CODESEGM    ENDS

;##################################################

   END
```

Analizzando il file **TESTLIB1.ASM** possiamo notare innanzi tutto che le macro alfanumeriche **riga** e **colonna** vengono prima dichiarate all'interno di **setCursor**, e poi ridichiarate all'interno di **printStr**; dall'inizio di **setCursor** sino all'inizio di **printStr** il corpo della macro **riga** vale **[bp+6]**, mentre da **printStr** in poi il corpo della macro **riga** vale **[bp+8]**. Se ci si dimentica di ridefinire la macro **riga**, all'interno di **printStr** l'assembler converte il nome **riga** in **[bp+6]** e il programma va in crash; ancora una volta quindi e' importante ribadire che in **Assembly**, la minima disattenzione da parte del programmatore puo' essere pagata a caro prezzo.
Un'altro aspetto importante da notare e' che la fine del modulo **TESTLIB1.ASM** viene delimitata dalla direttiva **END** senza nessun argomento aggiuntivo; un eventuale argomento verrebbe interpretato dall'assembler come l'identificatore di un entry point. In questo caso l'assembler produce un messaggio di errore per indicare che sono stati individuati due o piu' entry point; in sostanza, in un programma formato da due o piu' moduli, solo uno dei moduli (**modulo principale**) deve contenere l'entry point.

Torniamo per un momento al discorso relativo alla visibilita' degli identificatori; nel blocco **CODESEGM** del modulo **TESTLIB1.ASM** potremmo ad esempio definire una etichetta chiamata **start**. Questa etichetta non andrebbe in conflitto con l'etichetta **start** di **TESTMAIN.ASM** in quanto i due identificatori hanno entrambi visibilita' limitata ai moduli di appartenenza; e' chiaro che se si fa in modo che l'etichetta **start** di **TESTLIB1.ASM** diventi visibile nel modulo **TESTMAIN.ASM** si ottiene un messaggio di errore da parte dell'assembler. Ci si puo' chiedere come facciano l'assembler e il linker a non confondere tra loro questi due identificatori che oltre tutto si trovano nello stesso blocco **CODESEGM**; per ottenere una risposta a questa domanda e' sufficiente analizzare quello che succede nelle fasi di assembling e di linking del programma.

Ora che siamo in possesso dei due moduli **TESTMAIN.ASM** e **TESTLIB1.ASM** possiamo passare alla fase di assemblaggio; il comando da impartire con il **TASM** e':
`tasm testmain.asm + testlib1.asm.`
Il comando da impartire con il **MASM** e':
`ml /c testmain.asm testlib1.asm.`
L'ordine con il quale i moduli vengono passati all'assembler non ha nessuna importanza; se viene rilevato un errore in uno solo dei due moduli, e' sufficiente riassemblare solo quel modulo.
L'assemblaggio dei due blocchi **DATASEGM** e **STACKSEGM** del modulo **TESTMAIN.ASM** produce l'identico risultato visibile in Figura 2 e in Figura 4; l'assemblaggio del blocco **CODESEGM** del modulo **TESTMAIN.ASM** produce invece il risultato visibile in Figura 9.

Figura 9 - Assemblaggio di CODESEGM (modulo TESTMAIN.ASM)

```
0000h    CODESEGM    SEGMENT  PARA PUBLIC USE16       'CODE'

0000h    B8 0000s       ; mov       ax, DATASEGM
0003h    8E D8          ; mov       ds, ax

0005h    6A 0A          ; push      10
0007h    6A 00          ; push      0
0009h    68 0000r       ; push      offset stringa1
000Ch    E8 0000e       ; call      near ptr printStr

000Fh    6A 0B          ; push      11
0011h    6A 00          ; push      0
0013h    68 0020r       ; push      offset stringa2
0016h    E8 0000e       ; call      near ptr printStr

0019h    6A 0C          ; push      12
001Bh    6A 00          ; push      0
```

```
001Dh    68 0040r       ; push      offset stringa3
0020h    E8 0000e       ; call      near ptr printStr

0023h    B0 00          ; mov       al, 00h
0025h    B4 4C          ; mov       ah, 4ch
0027h    CD 21          ; int       21h

0029h    CODESEGM    ENDS
```

Come si vede in Figura 9, l'assembler e' in grado di determinare gli offset potenzialmente rilocabili (**r**) delle tre variabili **stringa1 stringa2** e **stringa3**; questo e' possibile in quanto queste tre variabili vengono definite nel modulo **TESTMAIN.ASM** appena assemblato. L'assembler non e' invece in grado di determinare l'offset di **printStr** in quanto questo identificatore non viene definito nel modulo **TESTMAIN.ASM**; grazie pero' alla direttiva **EXTRN** l'assembler capisce che questo identificatore e' una etichetta di tipo **NEAR** che viene definita in un altro modulo. Osserviamo infatti che nella chiamata di **printStr** l'assembler utilizza il codice macchina **E8h** (chiamata diretta intrasegmento) seguito dal valore simbolico **0000e**; la **e** significa proprio **extern** ed e' una informazione che l'assembler passa al linker. In fase di linking il linker provvedera' a modificare questo valore; tutta questa situazione viene riassunta dalla **Symbol Table** visibile in Figura 10.

Figura 10 - Symbol Table di TESTMAIN.ASM

```
Symbol Name        Type        Value

STACK_SIZE         Number      0400h
printStr           Near        CODESEGM:---- Extern
setCursor          Near        CODESEGM:---- Extern
start              Near        CODESEGM:0000h
stringa1           Byte        DATASEGM:0000h
stringa2           Byte        DATASEGM:0020h
stringa3           Byte        DATASEGM:0040h

Groups & Segments     Bit  Size    Align   Combine Class

CODESEGM              16   0029h   Para    Public  CODE
DATASEGM              16   0060h   Para    Public  DATA
STACKSEGM             16   0400h   Para    Stack   STACK
```

Un dato importante da tener presente e' la dimensione in byte del blocco **CODESEGM** del modulo **TESTMAIN.ASM**; come si vede in Figura 9 e in Figura 10, questo blocco richiede **29h** byte di memoria.

Passiamo ora al modulo **TESTLIB1.ASM**; l'assemblaggio del blocco **CODESEGM** di questo modulo produce il risultato visibile in Figura 11.

Figura 11 - Assemblaggio di CODESEGM (modulo TESTLIB1.ASM)

```
0000h    CODESEGM    SEGMENT  PARA PUBLIC USE16      'CODE'

0000h    55              ; push      bp
0001h    8B EC           ; mov       bp, sp

0003h    B4 02           ; mov       ah, 02h
0005h    B7 00           ; mov       bh, 00h
0007h    8A 76 06        ; mov       dh, riga
000Ah    8A 56 04        ; mov       dl, colonna
000Dh    CD 10           ; int       10h

000Fh    5D              ; pop       bp
0010h    C2 0004         ; ret       4

0013h    55              ; push      bp
0014h    8B EC           ; mov       bp, sp

0016h    FF 76 08        ; push      word ptr riga
0019h    FF 76 06        ; push      word ptr colonna
001Ch    E8 FFE1         ; call      near ptr setCursor

001Fh    8B 56 04        ; mov       dx, strAddr
0022h    B4 09           ; mov       ah, 09h
0024h    CD 21           ; int       21h

0026h    5D              ; pop       bp
0027h    C2 0006         ; ret       6

002Ah    CODESEGM    ENDS
```

In Figura 11 e' importante notare i codici macchina delle istruzioni che coinvolgono le due macro alfanumeriche **riga** e **colonna**; questi codici macchina cambiano anche in base alle varie ridichiarazioni delle macro.
Complessivamente il blocco **CODESEGM** del modulo **TESTLIB1.ASM** richiede **2Ah** byte di memoria; la Figura 12 mostra la **Symbol Table** prodotta dall'assembler per questo modulo.

Figura 12 - Symbol Table di TESTLIB1.ASM

```
Symbol Name        Type         Value

colonna            Text         [bp+6]
printStr           Near         CODESEGM:0013h
riga               Text         [bp+8]
setCursor          Near         CODESEGM:0000h
strAddr            Text         [bp+4]

Groups & Segments     Bit  Size    Align   Combine Class

CODESEGM              16   002Ah   Para    Public  CODE
```

Il valore assegnato alle macro alfanumeriche e' quello relativo all'ultima ridichiarazione.

A questo punto entra in azione il linker che questa volta deve eseguire un lavoro leggermente piu' complesso rispetto al primo esempio del capitolo; in particolare, nel caso di un programma formato da due o piu' moduli diventa importante anche l'ordine con il quale i moduli stessi vengono passati al linker. Cominciamo dal caso in cui i moduli vengano passati al linker nell'ordine piu' intuitivo, e cioe **TESTMAIN.OBJ** seguito poi da **TESTLIB1.OBJ**; il comando da impartire con il **TASM** e':

```
tlink testmain.obj + testlib1.obj.
```

Il comando da impartire con il **MASM** e':

```
link testmain.obj testlib1.obj.
```

Il lavoro svolto dal linker parte con l'analisi dei singoli segmenti di programma; il primo segmento incontrato dal linker e' il blocco **DATASEGM** di **TESTMAIN.OBJ**. Siccome non esiste nessun altro blocco **DATASEGM** ne in **TESTMAIN.OBJ** ne in **TESTLIB1.OBJ**, il linker produce il seguente risultato:

```
Start   Stop    Length  Name      Class

00000h  0005Fh  00060h  DATASEGM  DATA
```

Questo risultato e' identico al caso del blocco **DATASEGM** di Figura 1; naturalmente il linker non effettua nessuna rilocazione degli offset presenti in **DATASEGM**.
Il secondo segmento incontrato dal linker e' il blocco **CODESEGM** di **TESTMAIN.OBJ**; il linker trova anche in **TESTLIB1.OBJ** un blocco avente lo stesso nome lo stesso attributo di combinazione e la stessa classe di **CODESEGM**, e procede quindi alla fusione dei due blocchi. Il primo blocco **CODESEGM** occupa **29h** byte e richiede un allineamento al paragrafo; il linker produce quindi il seguente risultato:

```
Start   Stop    Length  Name      Class

00060h  00088h  00029h  CODESEGM  CODE
```

Il secondo blocco **CODESEGM** occupa **2Ah** byte e richiede un allineamento al paragrafo; il linker produce quindi il seguente risultato:

```
Start   Stop    Length  Name      Class

00090h  000B9h  0002Ah  CODESEGM  CODE
```

Unendo i due blocchi **CODESEGM** il linker ottiene un blocco unico **CODESEGM** che occupa tutti gli indirizzi fisici compresi tra **00060h** e **000B9h**; la lunghezza totale di **CODESEGM** e' quindi:

```
000B9h - 00060h + 1h = 0005Ah.
```

Come al solito il **+1** tiene conto del fatto che gli indici partono da zero; in definitiva, il blocco finale **CODESEGM** assume le seguenti caratteristiche:

```
Start   Stop    Length  Name      Class

00060h  000B9h  0005Ah  CODESEGM  CODE
```

A questo punto il linker deve procedere alla rilocazione di tutti gli offset presenti nella seconda parte del blocco **CODESEGM**; in Figura 9 vediamo che la prima parte del blocco **CODESEGM** occupa tutti gli indirizzi logici compresi tra **0000h:0000h** e **0000h:0029h**. Dopo **0000h:0029h**, il prossimo indirizzo logico allineato al paragrafo e' **0000h:0030h**; cio' significa che il linker deve sommare il valore **0030h** a tutti gli offset presenti nella seconda parte del blocco **CODESEGM** (Figura 11). Possiamo dire allora che l'offset iniziale **0000h** della procedura **setCursor** diventa (Figura 11):

```
0000h + 0030h = 0030h;
```

analogamente, l'offset iniziale **0013h** della procedura **printStr** diventa (Figura 11):

```
0013h + 0030h = 0043h.
```

Tutto cio' ci permette anche di rispondere alla domanda su come faccia il linker a non confondere due identificatori aventi lo stesso nome; supponiamo ad esempio di definire una seconda etichetta **start** all'offset **0000h** del blocco **CODESEGM** di **TESTLIB1.ASM**. Subito dopo la fusione dei due blocchi con la conseguente rilocazione di tutti gli offset del secondo blocco **CODESEGM**, le due etichette vengono a trovarsi in due offset differenti e quindi non possono essere confuse; osserviamo infatti che la **start** di **TESTMAIN.ASM** rimane all'offset **0000h**, mentre la **start** di **TESTLIB1.ASM** viene rilocata all'offset:

```
0000h + 0030h = 0030h.
```

Il lavoro svolto dal linker sul blocco finale **CODESEGM** non e' ancora terminato; rimangono infatti da risolvere le istruzioni di chiamata di **printStr** lasciate in sospeso dall'assembler. Consideriamo ad esempio l'istruzione:

```
call near ptr printStr,
```

che si trova all'offset **000Ch** del blocco **CODESEGM** di Figura 9; il codice macchina generato dall'assembler e':

```
E8 0000e.
```

Il linker trova il codice **E8h** e capisce che si tratta di una chiamata diretta intrasegmento; subito dopo il linker trova il codice **0000e**, e dalla **e** capisce che questo valore deve essere modificato. Come gia' sappiamo, il valore da inserire e' quello che sommato all'offset dell'istruzione successiva alla **CALL** (indirizzo di ritorno) ci fornisce l'offset della procedura da chiamare; il linker vede che l'indirizzo di ritorno e' **000Fh** (Figura 9), mentre l'offset rilocato di **printStr** e' **0043h**. Il valore da sostituire a **0000e** e' quindi:

```
0043h - 000Fh = 0034h;
```

di conseguenza il linker genera il codice macchina definitivo:

```
E8 0034.
```

Il terzo e ultimo segmento di programma incontrato dal linker e' il blocco **STACKSEGM** di **TESTMAIN.OBJ**, per il quale abbiamo richiesto un allineamento al paragrafo; siccome non esiste nessun altro blocco **STACKSEGM** ne in **TESTMAIN.OBJ** ne in **TESTLIB1.OBJ**, il linker produce il seguente risultato:

```
Start   Stop    Length  Name      Class

000C0h  004BFh  00400h  STACKSEGM STACK
```

Il risultato complessivo prodotto dal linker viene mostrato in Figura 13.

Figura 13 - Map File TESTMAIN.MAP

```
Start    Stop     Length   Name        Class

00000h   0005Fh   00060h   DATASEGM    DATA
00060h   000B9h   0005Ah   CODESEGM    CODE
000C0h   004BFh   00400h   STACKSEGM   STACK

Program entry point at 0006h:0000h
```

Notiamo subito che l'entry point calcolato dal linker e' identico a quello di Figura 6; questo fatto e' abbastanza ovvio in quanto, come si vede in Figura 13, il blocco **CODESEGM** parte dall'indirizzo fisico **00060h**, mentre l'etichetta **start** si trova all'offset **0000h** di **CODESEGM**. Il linker di conseguenza nell'**header** di **TESTMAIN.EXE** inserisce: **InitialCS=0006h** e **InitialIP=0000h**.

La situazione cambia radicalmente nel momento in cui viene invertito l'ordine di linkaggio dei due moduli **TESTMAIN.ASM** e **TESTLIB1.ASM**; proviamo infatti a chiamare il linker con il comando:

```
tlink testlib1.obj + testmain.obj.
```

Il nuovo **Map File** prodotto dal linker viene mostrato in Figura 14.

Figura 14 - Map File TESTLIB1.MAP

```
Start    Stop     Length   Name        Class

00000h   00058h   00059h   CODESEGM    CODE
00060h   000BFh   00060h   DATASEGM    DATA
000C0h   004BFh   00400h   STACKSEGM   STACK

Program entry point at 0000h:0030h
```

Prima di tutto si nota che il linker questa volta ha prodotto un **Map File** chiamato **TESTLIB1.MAP**, e un eseguibile chiamato **TESTLIB1.EXE**; cio' e' dovuto al fatto che il comportamento predefinito del linker consiste nell'assegnare all'eseguibile il nome del primo modulo della lista.
Il linker inizia quindi ad esaminare per primo il modulo **TESTLIB1.OBJ** dove trova il blocco **CODESEGM**; questo blocco occupa **2Ah** byte di memoria e richiede un allineamento al paragrafo, per cui si ottiene:

```
Start   Stop    Length  Name      Class

00000h  00029h  0002Ah  CODESEGM  CODE
```

Il linker trova anche in **TESTMAIN.OBJ** un blocco avente lo stesso nome lo stesso attributo di combinazione e la stessa classe di **CODESEGM**, e procede quindi alla fusione dei due blocchi. Il secondo blocco **CODESEGM** occupa **29h** byte e richiede un allineamento al paragrafo; il linker produce quindi il seguente risultato:

```
Start   Stop    Length  Name      Class

00030h  00058h  00029h  CODESEGM  CODE
```

Unendo i due blocchi **CODESEGM** il linker ottiene un blocco unico **CODESEGM** che occupa tutti gli indirizzi fisici compresi tra **00000h** e **00058h**; la lunghezza totale di **CODESEGM** e' quindi:

`00058h - 00000h + 1h = 00059h.`

Il **+1** tiene conto del fatto che gli indici partono da zero; in definitiva, il blocco finale **CODESEGM** assume le seguenti caratteristiche:

```
Start   Stop    Length  Name      Class

00000h  00058h  00059h  CODESEGM  CODE
```

A questo punto il linker deve procedere alla rilocazione di tutti gli offset presenti nella seconda parte del blocco **CODESEGM**; in Figura 11 vediamo che la prima parte del blocco **CODESEGM** occupa tutti gli indirizzi logici compresi tra **0000h:0000h** e **0000h:002Ah**. Dopo **0000h:002Ah**, il prossimo indirizzo logico allineato al paragrafo e' **0000h:0030h**; cio' significa che il linker deve sommare il valore **0030h** a tutti gli offset presenti nella seconda parte del blocco **CODESEGM** (Figura 9). Possiamo dire allora che l'offset iniziale **0000h** dell'etichetta **start** diventa (Figura 9):

`0000h + 0030h = 0030h.`

Terminato l'esame di **TESTLIB1.OBJ** il linker passa a **TESTMAIN.OBJ**; il primo blocco che il linker incontra in questo modulo e' **DATASEGM**. Questo blocco occupa **60h** byte e richiede un allineamento al paragrafo; siccome non esiste nessun altro blocco **DATASEGM** ne in **TESTMAIN.OBJ** ne in **TESTLIB1.OBJ**,il linker produce quindi il seguente risultato:

```
Start   Stop    Length  Name      Class

00060h  000BFh  00060h  DATASEGM  DATA
```

Il secondo blocco che il linker incontra in **TEASTMAIN.OBJ** e' **CODESEGM**; questo blocco e' gia' stato fuso con quello presente in **TESTLIB1.OBJ** per cui il linker passa direttamente al blocco successivo e cioe' a **STACKSEGM**. Il blocco **STACKSEGM** occupa **400h** byte e richiede un allineamento al paragrafo; siccome non esiste nessun altro blocco **STACKSEGM** ne in **TESTMAIN.OBJ** ne in **TESTLIB1.OBJ**, il linker produce il seguente risultato:

```
Start   Stop    Length  Name      Class

000C0h  004BFh  00400h  STACKSEGM STACK
```

Il risultato complessivo prodotto dal linker e' visibile nel **Map File** di Figura 14; l'inversione dell'ordine di linkaggio dei moduli ha prodotto un eseguibile nel quale la successione dei vari segmenti di programma e': **CODESEGM**, **DATASEGM**, **STACKSEGM**. Un'altra conseguenza e' data dal fatto che nel blocco **CODESEGM**, il codice macchina di **setCursor** e di **printStr** viene inserito prima dell'etichetta **start**; infatti, l'offset di **start** viene rilocato dal linker e diventa **0030h**.
Le considerazioni appena esposte giustificano il fatto che che questa volta l'entry point viene posto all'indirizzo logico **0000h:0030h** (Figura 14); infatti, in base al lavoro appena svolto dal linker, il blocco **CODESEGM** parte dall'indirizzo fisico **00000h**, mentre l'etichetta **start** si trova all'offset rilocato **0030h** di **CODESEGM**. Il linker di conseguenza nell'**header** di **TESTLIB1.EXE** inserisce: **InitialCS=0000h** e **InitialIP=0030h**.
Se abbiamo la necessita' assoluta di disporre i vari segmenti di programma in un determinato ordine, dobbiamo tener conto di tutti i concetti appena illustrati; piu' avanti viene mostrata una tecnica molto semplice che ci permette di costringere il linker a disporre i segmenti di programma come noi vogliamo.

Un'ultima considerazione riguarda l'eventualita' di dover distribuire su piu' moduli anche il blocco **DATASEGM** dell'esempio di Figura 7; supponiamo ad esempio di lasciare in un blocco **DATASEGM** del modulo **TESTMAIN.ASM** la definizione di **stringa1**, spostando invece in un blocco **DATASEGM** del modulo **TESTLIB1.ASM** le definizioni di **stringa2** e **stringa3**.
Applicando le cose dette in precedenza possiamo dire che il blocco **DATASEGM** di **TESTMAIN.ASM** assume la seguente struttura:

```
DATASEGM     SEGMENT  PARA PUBLIC USE16 'DATA'

   extrn     stringa2: byte, stringa3: byte

stringa1     db    'Stringa1 definita in DATASEGM', 10, 13, '$'

DATASEGM     ENDS
```

Il blocco **DATASEGM** di **TESTLIB1.ASM** assume invece la seguente struttura:

```
DATASEGM     SEGMENT  PARA PUBLIC USE16 'DATA'

   public    stringa2, stringa3

stringa2     db    'Stringa2 definita in DATASEGM', 10, 13, '$'
stringa3     db    'Stringa3 definita in DATASEGM', 10, 13, '$'

DATASEGM     ENDS
```

Programmi in formato COM distribuiti su piu' moduli.

Analizziamo ora il caso di un programma in formato **COM** che vogliamo distribuire su due o piu' moduli; in questo caso dobbiamo tener conto come sappiamo, di alcune limitazioni che caratterizzano questo tipo di eseguibili.
Un programma in formato **COM** e' costituito da un unico segmento di programma contenente codice, dati e stack; questo segmento di programma puo' essere suddiviso in tante parti, e ciascuna di queste parti puo' essere inserita in un modulo differente. In fase di linkaggio del programma il linker provvede a fondere queste parti tra loro ottenendo alla fine un segmento unico; l'importante e' che la dimensione complessiva di questo segmento unico non superi i **65536** byte. Naturalmente questa limitazione vale anche per i singoli segmenti di programma di un eseguibile in formato **EXE**.
L'aspetto piu' delicato per un programma in formato **COM** riguarda il fatto che in questo tipo di eseguibile, l'entry point deve trovarsi tassativamente all'offset **0100h** del segmento unico; tutto cio' significa che nella generazione di un eseguibile in formato **COM** diventa fondamentale l'ordine con il quale i vari moduli vengono passati al linker. Per chiarire questo aspetto vediamo un esempio pratico; in Figura 15 vediamo il modulo principale (semplificato) **COMFILE1.ASM** di un programma in formato **COM**.

Figura 15 - Modulo COMFILE1.ASM

```
.386

COMSEGM      SEGMENT  PARA PUBLIC USE16 'SEG_UNICO'

   extrn     printStr: near
   extrn     stringa2: byte, stringa3: byte

   assume    cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM

   org       0100h

start:

   push      offset stringa1
   call      printStr

   push      offset stringa2
   call      printStr

   push      offset stringa3
   call      printStr

   mov       al, 00h
   mov       ah, 4Ch
   int       21h

stringa1     db        'Stringa n. 1', 10, 13, '$'

COMSEGM      ENDS

   END       start
```

Come si puo' notare, il modulo principale **COMFILE1.ASM** contiene non solo l'entry point del programma, ma anche la direttiva **ORG**; questa direttiva incrementa di **256** byte il location counter **$** in modo da creare lo spazio per il **PSP** proprio all'inizio di **COMSEGM**. La direttiva **start** si viene a trovare di conseguenza all'offset **0100h** di **COMSEGM**.

E' importante anche ricordare che in un eseguibile in formato **COM** il programmatore non deve inizializzare nessun registro di segmento in quanto questo lavoro spetta al **SO**; nel caso del nostro esempio, il **SO** carica il programma in memoria ponendo: **CS = DS = ES = SS = COMSEGM**.

In Figura 16 vediamo il modulo **COMFILE2.ASM** del programma.

Figura 16 - Modulo COMFILE2.ASM

```
COMSEGM     SEGMENT  PARA PUBLIC USE16 'SEG_UNICO'

   public   stringa2

   assume   cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM

stringa2    db       'Stringa n. 2', 10, 13, '$'

COMSEGM     ENDS

   END
```

Il modulo **COMFILE2.ASM** non deve contenere nessun entry point, ma e' libero di utilizzare eventualmente la direttiva **ORG** per modificare il location counter **$** all'interno di **COMSEGM**.
In Figura 17 vediamo il modulo **COMFILE3.ASM** del programma.

Figura 17 - Modulo COMFILE3.ASM

```
COMSEGM     SEGMENT  PARA PUBLIC USE16 'SEG_UNICO'

   public   printStr, stringa3

   assume   cs: COMSEGM, ds: COMSEGM, es: COMSEGM, ss: COMSEGM

; procedure printStr(var strAddr: String)

printStr proc near

   strAddr  equ   [bp+4]

   push     bp                     ; preserva bp
   mov      bp, sp                 ; ss:bp = ss:bp

   mov      dx, strAddr            ; ds:dx punta a strAddr
   mov      ah, 09h                ; servizio Display String
   int      21h                    ; chiama i servizi DOS

   pop      bp                     ; ripristina bp
   ret      2                      ; return + pulizia stack

printStr endp

stringa3    db       'Stringa n. 3', 10, 13, '$'

COMSEGM     ENDS

   END
```

Per il modulo **COMFILE3.ASM** e per altri eventuali moduli valgono le stesse considerazioni gia' esposte per **COMFILE2**.

A questo punto possiamo passare alla fase di assemblaggio; come gia' sappiamo, in questa fase l'ordine con il quale passiamo i vari moduli all'assembler non ha nessuna importanza. Possiamo impartire ad esempio il comando:

```
tasm comfile3.asm + comfile2.asm + comfile1.asm.
```

In questo modo otteniamo (se non vengono trovati errori) i tre object files **COMFILE1.OBJ**, **COMFILE2.OBJ** e **COMFILE3.OBJ**; questi tre files devono essere passati al linker per ottenere l'eseguibile finale in formato **COM**. Nella fase di linking diventa pero' importantissimo l'ordine con il quale i vari moduli vengono passati al linker; come si puo' facilmente immaginare, siamo obbligati a passare per primo il modulo principale **COMFILE1.OBJ** contenente l'entry point all'offset **0100h**. Il comando corretto da impartire e' quindi:

```
tlink /t comfile1.obj + comfile2.obj + comfile3.obj,
```

oppure:

```
tlink /t comfile1.obj + comfile3.obj + comfile2.obj.
```

Se proviamo a passare per primo **COMFILE2.OBJ** o **COMFILE3.OBJ** otteniamo ovviamente un errore del linker; e' chiaro infatti che in un caso del genere il blocco **COMSEGM** contenuto in **COMFILE2.OBJ** e/o **COMFILE3.OBJ** verrebbe a precedere nell'eseguibile il blocco **COMSEGM** contenuto in **COMFILE1.OBJ**. Di conseguenza il blocco **COMSEGM** contenuto in **COMFILE1.OBJ** subirebbe la rilocazione di tutti i suoi offset; come risultato finale l'entry point si verrebbe a trovare ad un offset non valido in quanto (sicuramente) superiore a **0100h**.
Nota.
Il **MASM** permette di creare un eseguibile in formato **COM** con entry point all'offset **0000h**; naturalmente se si prova ad eseguire un programma del genere si ottengono risultati imprevedibili.

Caso generale.

Il caso piu' generale possibile si presenta quando vogliamo distribuire su piu' moduli un programma dotato di piu' segmenti di dati e piu' segmenti di codice; in una situazione del genere conviene assegnare ad uno dei moduli il ruolo di **modulo principale**. Il modulo piu' adatto per ricoprire questo ruolo e' quello che contiene l'entry point e quindi anche il blocco principale delle istruzioni; oltre all'entry point, il modulo principale e' in genere anche quello che fornisce lo stack all'intero programma. Nei moduli secondari vengono distribuite invece le varie procedure utilizzate dal modulo principale; ciascuno dei moduli secondari puo' essere visto quindi come una libreria di procedure da linkare al modulo principale. In genere ogni libreria e' dotata di un proprio segmento di codice e di un proprio segmento dati; il segmento dati contiene principalmente variabili legate a necessita' interne della libreria (variabili private), ma anche variabili visibili dal modulo principale (variabili pubbliche o globali). Nel caso piu' generale i segmenti di codice e di dati di una libreria hanno attributi (compreso il nome) diversi da quelli degli analoghi segmenti usati dal modulo principale; di conseguenza il modulo principale ha bisogno di conoscere tutti questi attributi per potersi collegare ai moduli secondari. Proprio per questo motivo, ogni libreria e' sempre accompagata da una adeguata documentazione; il compito della documentazione e' quello di fornire informazioni dettagliate sugli attributi dei segmenti di programma usati dalla libreria, e sulle modalita' di chiamata (interfaccia) delle procedure della libreria stessa. In una situazione di questo genere, il modulo principale per poter chiamare le procedure di una libreria deve utilizzare chiamate di tipo **FAR**; analogamente, se il modulo principale vuole utilizzare le eventuali variabili globali fornite da una libreria, deve conoscere non solo l'offset ma anche il segmento di appartenenza di queste variabili.
Per illustrare in pratica tutti questi concetti, riscriviamo l'esempio mostrato nelle figure 7 e 8; questa volta utilizziamo le convenzioni del linguaggio **C** per il passaggio degli argomenti e per la pulizia dello stack. Prima di tutto scriviamo il codice sorgente del modulo secondario chiamato **FILELIB1.ASM**; questo modulo contiene la procedura **setCursor** che puo' essere chiamata se necessario anche da procedure appartenenti ad altre librerie. Nel modulo **FILELIB1.ASM** e'

presente anche un blocco dati contenente variabili private destinate all'uso interno della libreria; il modulo **FILELIB1.ASM** contenente la procedura **setCursor** viene mostrato in Figura 18.

Figura 18 - Modulo FILELIB1.ASM

```
;--------------------------------------------------;
; File filelib1.asm                                ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                           ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

;############ segmento dati privati ###############

LIB1DATA    SEGMENT  PARA PRIVATE USE16 'DATA'

varLib1a    dw       1
varLib1b    dw       2

LIB1DATA    ENDS

;############### segmento codice ##################

LIB1CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

   public   setCursor

   assume   cs: LIB1CODE            ; assegna LIB1CODE a CS

;------------- inizio blocco procedure ------------

; void far setCursor(int riga, int colonna)

setCursor proc far

   riga     equ   [bp+6]            ; parametro riga
   colonna  equ   [bp+8]            ; parametro colonna

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:bp

   mov      ah, 02h                 ; ah = Set Cursor Position
   mov      bh, 00h                 ; bh = pagina video 0
   mov      dh, riga                ; dh = riga
   mov      dl, colonna             ; dl = colonna
   int      10h                     ; chiama i servizi Video BIOS

   pop      bp                      ; ripristina bp
   ret                              ; far return

setCursor endp

;-------------- fine blocco procedure -------------

LIB1CODE    ENDS

;##################################################

   END
```

Il modulo **FILELIB1.ASM** utilizza un blocco dati **LIB1DATA** contenente variabili che non sono visibili all'esterno del file in quanto sono riservate all'uso esclusivo della libreria; l'attributo di combinazione **PRIVATE** impedisce che il blocco **LIB1DATA** venga fuso con altri blocchi compatibili.
La procedura **setCursor** questa volta segue le convenzioni **C**; di conseguenza i parametri si trovano disposti nello stack nello stesso ordine indicato dal prototipo. Si puo' anche notare che a causa della **FAR** call, il primo parametro si trova a **[bp+6]**.
Un aspetto molto delicato riguarda l'eventualita' che le procedure definite nel modulo **FILELIB1.ASM** debbano accedere per nome alle variabili private del modulo stesso; in un caso del genere queste procedure devono modificare necessariamente **DS** o **ES** ponendo ad esempio **DS=LIB1DATA**. E' fondamentale che tutte le procedure che modificano **DS** (o altri registri di segmento) ne preservino il contenuto originario; in caso contrario il caller dopo aver riottenuto il controllo rischierebbe di mandare in crash il programma.

In Figura 19 viene mostrato il modulo secondario **FILELIB2.ASM** che contiene la procedura **printStr**; questa procedura si serve di **setCursor** per posizionare il cursore sullo schermo.

Figura 19 - Modulo FILELIB2.ASM

```
;--------------------------------------------------;
; File filelib2.asm                                ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                            ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

;################ segmento dati ###################

LIB2DATA    SEGMENT  PARA PUBLIC USE16 'DATA'

   public   strLib2b, strLib2c

strLib2a    db       'strLib2a definita in LIB2DATA', 10, 13, '$'
strLib2b    db       'strLib2b definita in LIB2DATA', 10, 13, '$'
strLib2c    db       'strLib2c definita in LIB2DATA', 10, 13, '$'

LIB2DATA    ENDS

;########## segmento codice di FILELIB1 ###########

LIB1CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

   extrn    setCursor: far

LIB1CODE    ENDS

;############### segmento codice #################

LIB2CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

   public   printStr

   assume   cs: LIB2CODE            ; assegna LIB2CODE a CS

;------------- inizio blocco procedure ------------
```

```
; void far printStr(int riga, int colonna, char far *strAddr)
; strAddr = strSeg + strOffs

printStr proc far

   riga     equ   [bp+6]            ; parametro riga
   colonna  equ   [bp+8]            ; parametro colonna
   strOffs  equ   [bp+10]           ; parametro strOffs
   strSeg   equ   [bp+12]           ; parametro strSeg

   push     bp                      ; preserva bp
   mov      bp, sp                  ; ss:bp = ss:bp
   push     ds                      ; preserva ds

   push     word ptr colonna        ; passa colonna
   push     word ptr riga           ; passa riga
   call     far ptr setCursor       ; chiama setCursor
   add      sp, 4                   ; pulisce lo stack

   mov      ds, strSeg              ; ds = seg(stringa)
   mov      dx, strOffs             ; ds:dx punta a strAddr
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS

   pop      ds                      ; ripristina ds
   pop      bp                      ; ripristina bp
   ret                              ; far return

printStr endp

;------------- fine blocco procedure ------------

LIB2CODE    ENDS

;###################################################

   END
```

Il modulo **FILELIB2.ASM** contiene un blocco dati **LIB2DATA** all'interno del quale vengono definite le tre stringhe **strLib2a**, **strLib2b** e **strLib2c**; solo la seconda e la terza stringa sono visibili all'esterno in quanto sono state dichiarate **PUBLIC**. Le procedure di **FILELIB2.ASM** che vogliono accedere per nome alle variabili definite in **LIB2DATA**, devono seguire le stesse precauzioni gia' esposte per **setCursor**; in altre parole, tutte le procedure che modificano ad esempio **DS**, ne devono preservare il contenuto originario.
All'interno della procedura **printStr** notiamo la chiamata in stile **C** di **setCursor**; infatti i parametri vengono inseriti nello stack a partire dall'ultimo, mentre la pulizia dello stack spetta al caller e cioe' a **printStr**.
Per poter chiamare correttamente **setCursor**, la procedura **printStr** deve conoscere sia il segmento di definizione della stessa **setCursor**, sia il tipo di chiamata necessario (**NEAR** o **FAR**); a tale proposito notiamo che prima di **LIB2CODE** e' stato inserito un blocco **LIB1CODE** contenente unicamente la direttiva **EXTRN** per **setCursor**. Le direttive non occupano memoria, per cui la dimensione di questo blocco **LIB1CODE** e' pari a zero byte; il blocco **LIB1CODE** di **FILELIB2.ASM** viene fuso dal linker con il blocco **LIB1CODE** di **FILELIB1.ASM** senza nessuna conseguenza sulle dimensioni finali.
Quando si scrive un programma multisegmento distribuito su piu' moduli, e' fondamentale seguire le regole appena esposte per il collegamento degli identificatori esterni; in caso contrario si ottiene un programma eseguibile che (nella migliore delle ipotesi) funziona in modo anomalo.

La Figura 20 mostra infine il modulo principale del programma; questo modulo prende il nome di **FILEMAIN.ASM** e contiene lo stack, l'entry point e il blocco delle istruzioni principali.

Figura 20 - Modulo FILEMAIN.ASM

```
;--------------------------------------------------;
; File filemain.asm                                ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                                ; set di istruzioni a 32 bit

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h                  ; 1024 byte per lo stack

;################ segmento dati ###################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

strMain1    db    'strMain1 definita in DATASEGM', 10, 13, '$'
strMain2    db    'strMain2 definita in DATASEGM', 10, 13, '$'

DATASEGM    ENDS

;########### segmento dati di FILELIB2 ############

LIB2DATA    SEGMENT  PARA PUBLIC USE16 'DATA'

   extrn    strLib2b: byte, strLib2c: byte

LIB2DATA    ENDS

;########## segmento codice di FILELIB1 ###########

LIB1CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

   extrn    setCursor: far

LIB1CODE    ENDS

;########## segmento codice di FILELIB2 ###########

LIB2CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

   extrn    printStr: far

LIB2CODE    ENDS

;############### segmento codice ##################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM            ; assegna CODESEGM a CS

start:                              ; entry point

   mov      ax, DATASEGM            ; trasferisce DATASEGM
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS
```

```
;------- inizio blocco principale istruzioni ------

   push     seg strMain1            ; passa seg(strMain1)
   push     offset strMain1         ; passa offset(strMain1)
   push     0                       ; colonna = 0
   push     10                      ; riga = 10
   call     far ptr printStr        ; chiama printStr
   add      sp, 8                   ; pulisce lo stack

   push     seg strMain2            ; passa seg(strMain2)
   push     offset strMain2         ; passa offset(strMain2)
   push     0                       ; colonna = 0
   push     12                      ; riga = 12
   call     far ptr printStr        ; chiama printStr
   add      sp, 8                   ; pulisce lo stack

   push     seg strLib2b            ; passa seg(strLib2b)
   push     offset strLib2b         ; passa offset(strLib2b)
   push     0                       ; colonna = 0
   push     14                      ; riga = 14
   call     far ptr printStr        ; chiama printStr
   add      sp, 8                   ; pulisce lo stack

   push     seg strLib2c            ; passa seg(strLib2c)
   push     offset strLib2c         ; passa offset(strLib2c)
   push     0                       ; colonna = 0
   push     16                      ; riga = 16
   call     far ptr printStr        ; chiama printStr
   add      sp, 8                   ; pulisce lo stack

;-------- fine blocco principale istruzioni -------

   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; servizio Terminate Process
   int      21h                     ; chiama i servizi DOS

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)   ; 1024 byte per lo stack

STACKSEGM   ENDS

;##################################################

   END      start
```

Il modulo **FILEMAIN.ASM** utilizza le variabili definite in **LIB2DATA**, le procedure definite in **LIB1CODE** e le procedure definite in **LIB2CODE**; di conseguenza **FILEMAIN.ASM** deve conoscere sia i segmenti di definizione, sia le modalita' di accesso relative a tutti questi identificatori esterni. Come si puo' notare infatti, all'interno di **FILEMAIN.ASM** sono presenti i blocchi **LIB2DATA**, **LIB1CODE** e **LIB2CODE** contenenti le necessarie direttive **EXTRN**; se non si segue questo schema si ottiene un programma malfunzionante in quanto gli operatori **SEG** e **OFFSET** non sono in grado di operare correttamente.
Siccome stiamo lavorando esclusivamente con gli indirizzi logici **seg:offset** delle stringhe da passare a **printStr**, l'inizializzazione **DS=DATASEGM** e' superflua; infatti gli operatori **SEG** e

OFFSET fanno riferimento direttamente al segmento di appartenenza del loro operando.
Nel blocco codice principale viene chiamata la procedura **printStr** per visualizzare stringhe **DOS** definite in diversi segmenti di dati; proprio per questo motivo **printStr** ha bisogno dell'indirizzo completo **seg:offset** della stringa da visualizzare. Come al solito, e' importante ricordare che se si deve passare una coppia **seg:offset** ad una procedura, la componente **offset** deve essere inserita nello stack subito dopo la componente **seg**; in questo modo la coppia **seg:offset** viene disposta nello stack in modo che la componente **offset** preceda la componente **seg** nel rispetto della convenzione seguita dalle CPU **80x86**.

Nota importante.
Puo' anche capitare che il programmatore non conosca i segmenti di programma delle librerie in cui vengono definiti gli identificatori esterni da utilizzare nel modulo principale; questa situazione si presenta ad esempio quando abbiamo a disposizione librerie in formato **object file**, accompagnate da una documentazione che elenca solo l'interfaccia degli identificatori esterni. In un caso del genere e' fondamentale che nel modulo principale, tutte le direttive **EXTRN** vengano inserite al di fuori di qualsiasi segmento di programma; un punto adatto per l'inserimento di queste direttive e' ad esempio la sezione del modulo principale riservata alle direttive per l'assembler.
Supponiamo ad esempio di disporre delle due librerie **FILELIB1.OBJ** e **FILELIB2.OBJ** solo in formato **object file**; supponiamo poi che nella documentazione di queste due librerie siano presenti le seguenti informazioni in versione **C**:

```
extern char    strLib2b[];
extern char    strLib2c[];

extern void far setCursor(int riga, int colonna);
extern void far printStr(int riga, int colonna, char far *strAddr);
```

Per poter utilizzare questi identificatori esterni nel file **FILEMAIN.ASM**, non conoscendo il loro segmento di appartenenza dobbiamo inserire le necessarie direttive **EXTRN** al di fuori di qualsiasi segmento di programma; utilizzando la sezione riservata alle direttive per l'assembler, possiamo modificare la parte iniziale del modulo **TESTMAIN.ASM** in questo modo:

```
;--------------------------------------------------;
; File filemain.asm                                ;
;--------------------------------------------------;

;########### direttive per l'assembler ############

.386                               ; set di istruzioni a 32 bit

   EXTRN    strLib2b: BYTE, strLib2c: BYTE
   EXTRN    setCursor: FAR, printStr: FAR

;######### dichiarazione tipi e costanti ##########
```

Se ora proviamo ad assemblare **TESTMAIN.ASM**, viene generato anche il listing file **TESTMAIN.LST** contenente il codice macchina e la **Symbol Table** del modulo principale; se consultiamo proprio la **Symbol Table**, possiamo notare che nel caso ad esempio dell'identificatore **setCursor** l'assembler ha inserito le seguenti informazioni:
`setCursor Far ----:---- Extern.`
Come si puo' notare, l'assembler questa volta delega al linker il compito di individuare non solo l'offset di **setCursor**, ma anche il segmento di appartenenza; in fase di linking del programma, il linker provvedera' a sostituire il simbolo **----:----** con la coppia **seg:offset** di **setCursor**.

I files di inclusione (INCLUDE files).
In fase di linking del programma illustrato nelle figure 18, 19 e 20, l'ordine con il quale i tre object files **FILEMAIN.OBJ**, **FILELIB1.OBJ** e **FILELIB2.OBJ** vengono passati al linker non ha nessuna importanza; questo perche' stiamo generando un eseguibile in formato **EXE**. Il discorso cambia se abbiamo la necessita' di ottenere un eseguibile con i segmenti di programma disposti in un ben preciso ordine; supponiamo ad esempio di impartire il comando:

```
tlink filemain.obj + filelib1.obj + filelib2.obj.
```

In questo caso il linker produce un eseguibile chiamato **FILEMAIN.EXE**, ed un **Map File** chiamato **FILEMAIN.MAP**; la Figura 21 mostra appunto il **Map File** del nostro programma.

Figura 21 - Map File FILEMAIN.MAP

```
Start    Stop     Length   Name        Class

00000h   0003Fh   00040h   DATASEGM    DATA
00040h   0009Fh   00060h   LIB2DATA    DATA
000A0h   000A3h   00004h   LIB1DATA    DATA
000B0h   000CFh   00020h   LIB1CODE    CODE
000D0h   000EEh   0001Fh   LIB2CODE    CODE
000F0h   0013Dh   0004Eh   CODESEGM    CODE
00140h   0053Fh   00400h   STACKSEGM   STACK

Program entry point at 000Fh:0000h
```

Nella Figura 21 e' importante notare in particolare il raggruppamento dei segmenti effettuato dal linker in base all'attributo di classe.
Questa disposizione dei segmenti di programma, anche se non e' obbligatoria, e' sicuramente la piu' razionale possibile; osserviamo infatti che l'ordine con cui sono disposti i vari segmenti permette di soddisfare le dipendenze tra gli identificatori e le istruzioni che fanno riferimento ad essi. In sostanza, tutte le istruzioni di **CODESEGM** che fanno riferimento a identificatori (variabili e procedure) definiti in altri blocchi, vengono precedute dai prototipi degli identificatori stessi; in questo modo si facilita notevolmente il lavoro che l'assembler deve compiere per generare il codice macchina piu' opportuno. Si puo' notare inoltre che **LIB1CODE** precede **LIB2CODE**; anche questa disposizione e' la piu' razionale in quanto **LIB1CODE** contiene **setCursor** che viene usata da **printStr** in **LIB2CODE**.
La disposizione dei segmenti di programma diventa importantissima nel momento in cui abbiamo la necessita' di calcolare lo spazio in byte che il nostro programma occupa in memoria; osservando ad esempio la Figura 21, possiamo dire che il **Program Segment** di **FILEMAIN.EXE** richiede come minimo un blocco di memoria da:
`((STACKSEGM * 10h) + 0400h) - ((PSP * 10h) + 0000h)` byte.
Tutto cio' e' corretto solo se **STACKSEGM** e' l'ultimo segmento del nostro programma; in caso contrario si ottiene un calcolo privo di senso.
L'assembler fornisce diversi metodi che ci permettono di ordinare nel modo desiderato i vari segmenti di un programma; uno di questi metodi consiste nell'uso della direttiva **.ALPHA** per la disposizione in ordine alfabetico dei segmenti di programma. In un caso del genere, se vogliamo fare in modo che il blocco stack sia l'ultimo segmento del nostro programma, dovremmo attribuirgli un nome del tipo **ZZZZSTACK**; naturalmente bisogna anche fare in modo che non esistano altri nomi di segmenti alfabeticamente successivi a **ZZZZSTACK**.
Un'altro metodo consiste nell'uso della direttiva **.SEQ** per la disposizione in ordine sequenziale dei segmenti di programma; in assenza di diverse indicazioni da parte del programmatore, l'assembler e il linker utilizzano **.SEQ** come direttiva predefinita.
Sicuramente il metodo piu' potente e sofisticato e' quello che prevede l'uso combinato della direttiva **.SEQ** e degli **include files** (files di inclusione); generalmente ogni libreria di procedure e' accompagnata da un include file che e' un file contenente semplicemente tutte le direttive **EXTRN**

degli identificatori definiti nella libreria stessa. Un modulo principale che vuole utilizzare una determinata libreria, puo' servirsi del relativo include file attraverso la direttiva **INCLUDE**; in questo modo si evita di dover riscrivere ogni volta tutte le direttive **EXTRN** necessarie al modulo principale. Convenzionalmente gli include files utilizzano nomi con l'estensione predefinita **INC**; la Figura 22 mostra ad esempio l'include file **FILELIB1.INC** relativo alla libreria **FILELIB1.ASM** di Figura 18.

Figura 22 - FILELIB1.INC

```
LIB1CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

   extrn    setCursor: far

LIB1CODE    ENDS
```

La Figura 23 illustra invece l'include file **FILELIB2.INC** relativo alla libreria **FILELIB2.ASM** di Figura 19.

Figura 23 - FILELIB2.INC

```
LIB2DATA    SEGMENT  PARA PUBLIC USE16 'DATA'

   extrn    strLib2b: byte, strLib2c: byte

LIB2DATA    ENDS

LIB2CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

   extrn    printStr: far

LIB2CODE    ENDS
```

A questo punto, nel blocco delle direttive per l'assembler del modulo principale **FILEMAIN.ASM** possiamo scrivere:

```
; ######### direttive per l'assembler #########

.386                       ; set di istruzioni a 32 bit

INCLUDE  FILELIB1.INC      ; dichiarazioni per FILELIB1
INCLUDE  FILELIB2.INC      ; dichiarazioni per FILELIB2
```

Se proviamo ora a generare l'eseguibile con il comando:

```
tlink filemain.obj + filelib1.obj + filelib2.obj,
```

otteniamo il seguente **Map File**:

Figura 24 - Map File FILEMAIN.MAP

```
Start    Stop     Length   Name        Class

00000h   0001Fh   00020h   LIB1CODE    CODE
00020h   0003Eh   0001Fh   LIB2CODE    CODE
00040h   0008Dh   0004Eh   CODESEGM    CODE
00090h   000EFh   00060h   LIB2DATA    DATA
000F0h   0012Fh   00040h   DATASEGM    DATA
00130h   00133h   00004h   LIB1DATA    DATA
00140h   0053Fh   00400h   STACKSEGM   STACK

Program entry point at 0004h0000h
```

Questa situazione e' dovuta al fatto che il linker parte dal file **FILELIB1.INC**; in questo file viene incontrato il blocco **LIB1CODE**. A questo punto il linker dispone in sequenza tutti gli altri blocchi di codice; il secondo blocco di codice incontrato dal linker e' **LIB2CODE** in **FILELIB2.INC**, mentre il terzo blocco e' **CODESEGM** in **FILEMAIN.OBJ**.
Per quanto riguarda i blocchi di dati, il linker incontra per primo **LIB2DATA** in **FILELIB2.INC**; successivamente viene trovato **DATASEGM** in **FILEMAIN.OBJ** e **LIB1DATA** in **FILELIB1.OBJ**. L'ultimo segmento incontrato dal linker e' **STACKSEGM** in **FILEMAIN.OBJ**
Questa disposizione dei segmenti di programma viene chiaramente influenzata dalla struttura dei due include files; anche il punto di **FILEMAIN.ASM** nel quale inseriamo le direttive **INCLUDE** condiziona l'ordine dei segmenti di programma.
Se vogliamo riottenere la situazione di Figura 21 possiamo ricorrere ad un espediente molto semplice; dalle cose appena dette si intuisce infatti che e' possibile definire un ulteriore include file nel quale possiamo specificare la disposizione desiderata per tutti i segmenti di programma. La struttura di questo include file chiamato **ORDSEGM.INC** viene illustrata in Figura 25.

Figura 25 - ORDSEGM.INC

```
DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

DATASEGM    ENDS

LIB2DATA    SEGMENT  PARA PUBLIC USE16 'DATA'

LIB2DATA    ENDS

LIB1CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

LIB1CODE    ENDS

LIB2CODE    SEGMENT  PARA PUBLIC USE16 'CODE'

LIB2CODE    ENDS

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

CODESEGM    ENDS

STACKSEGM   SEGMENT  PARA STACK USE16 'CODE'

STACKSEGM   ENDS
```

Come si puo' notare, il file **ORDSEGM.INC** contiene una serie di segmenti vuoti che hanno il solo scopo di costringere il linker a seguire la disposizione da noi stabilita; il blocco **LIB1DATA** non e' presente in quanto e' riservato a **FILELIB1** e non e' quindi accessibile da **FILEMAIN.ASM** (che potrebbe anche ignorarne l'esitenza). Il file **ORDSEGM.INC** deve essere incluso per primo nel blocco delle direttive per l'assembler del file **FILEMAIN.ASM**; in pratica dobbiamo scrivere:

```
; ######### direttive per l'assembler #########

.386                       ; set di istruzioni a 32 bit

INCLUDE   ORDSEGM.INC      ; ordine segmenti
INCLUDE   FILELIB1.INC     ; dichiarazioni per FILELIB1
INCLUDE   FILELIB2.INC     ; dichiarazioni per FILELIB2
```

Se ora proviamo a generare l'eseguibile, possiamo constatare che il linker ha prodotto di nuovo la situazione mostrata in Figura 21.

Capitolo 29 - Caratteristiche avanzate di MASM e TASM

In un precedente capitolo e' stato detto che i linguaggi di programmazione di alto livello possono essere suddivisi fondamentalmente in linguaggi **interpretati** e linguaggi **compilati**; parallelamente, gli strumenti software che traducono in codice macchina i programmi scritti con un linguaggio di alto livello, si suddividono in **compilatori** e **interpreti**.

Gli interpreti svolgono questo lavoro di traduzione operando su una sola istruzione per volta (esecuzione **step by step** o passo passo); in sostanza, un interprete legge una istruzione del codice sorgente, la traduce in codice macchina e la esegue prima di passare all'istruzione successiva. Come si puo' facilmente intuire, nel caso dei linguaggi interpretati non viene creato nessun programma **eseguibile**; l'interprete infatti si serve esclusivamente del codice sorgente. Ogni singola istruzione del codice sorgente viene letta e tradotta in codice macchina; a questo punto l'interprete carica in **CS:IP** l'indirizzo del codice macchina appena ottenuto e cede il controllo alla CPU che provvede ad eseguire l'istruzione. Naturalmente, se l'istruzione contiene un riferimento a qualche dato in memoria, l'interprete ha il compito di comunicare alla CPU l'indirizzo del dato stesso; analogamente, se l'istruzione da eseguire prevede un salto ad un sottoprogramma, l'interprete comunica alla CPU l'indirizzo relativo al sottoprogramma stesso.
Queste considerazioni ci permettono di dedurre gli aspetti piu' rilevanti della programmazione con i linguaggi interpretati; la prima cosa di dire riguarda il fatto che se vogliamo eseguire un programma scritto con un linguaggio interpretato, e' indispensabile la presenza in memoria anche del relativo interprete. Tutta la complessita' del programma e' nascosta all'interno dell'interprete stesso che e' in grado non solo di effettuare la traduzione del codice sorgente in codice macchina, ma anche di fornire alla CPU tutte le informazioni necessarie per l'esecuzione di ogni singola istruzione; tutta la fase di traduzione ed esecuzione passo passo, si svolge chiaramente in modo estremamente lento. Il programma da eseguire e' presente sotto forma di codice sorgente per cui la sua struttura e' estremamente semplice e compatta; naturalmente, il codice sorgente non essendo "criptato" puo' essere letto da chiunque, anche all'insaputa del programmatore.

I compilatori invece svolgono il lavoro di traduzione operando sull'intero codice sorgente; questo significa che un compilatore legge un intero programma scritto con un linguaggio di alto livello e lo traduce in codice macchina in modo da ottenere un cosiddetto programma eseguibile. L'eseguibile cosi' ottenuto puo' essere caricato interamente in memoria ed eseguito quindi ad una velocita' enormemente superiore rispetto al caso dei programmi scritti con i linguaggi interpretati; il prezzo da pagare e' rappresentato dalla maggiore complessita' strutturale che si ripercuote sulle dimensioni dell'eseguibile stesso. Questa complessita' e' dovuta ovviamente al fatto che un programma eseguibile deve contenere al suo interno tutte le informazione necessarie al **SO** por poter caricare il programma stesso in memoria e per farlo eseguire dalla CPU. Tra le varie informazioni importanti si possono citare i valori iniziali da assegnare ai registri come **CS**, **DS**, **SS**, **SP** e **IP**; altre informazioni riguardano la quantita' di memoria richiesta per lo stack, la quantita' di memoria richiesta per i dati statici del programma, etc.
Per gestire tutta questa situazione, sono necessari due strumenti che come gia' sappiamo sono il **compilatore** e il **linker**; nel caso generale di un programma formato da due o piu' moduli, la generazione dell'eseguibile comporta quindi la fase di compilazione e la fase di linking. Nella fase di compilazione, ogni modulo sorgente viene convertito in un cosiddetto **modulo oggetto**; all'interno del modulo oggetto e' presente il codice macchina del modulo appena compilato, e la **symbol table** contenente la descrizione di ogni identificatore interno o esterno. Nella fase di linking i vari moduli oggetto vengono collegati tra loro in modo da ottenere l'eseguibile finale; in questa fase il compito piu' importante svolto dal linker consiste come sappiamo nel risolvere tutti gli eventuali riferimenti a identificatori esterni. Questa situazione si presenta ad esempio quando una istruzione presente in un modulo oggetto contiene un riferimento ad un dato o ad un

sottoprogramma definiti in un altro modulo oggetto; in un caso del genere il compilatore delega al linker il compito di calcolare gli indirizzi di questi identificatori. Al termine di tutte queste fasi, si ottiene un programma eseguibile del tutto autosufficiente che quindi, anche in assenza del compilatore, puo' girare su qualunque computer compatibile; osserviamo anche che i files eseguibili e i moduli oggetto, risultano illeggibili con un editor ASCII in quanto contengono informazioni in codice macchina che possono essere decriptate solo da un hacker esperto.

Le considerazioni appena esposte in relazione agli interpreti e ai compilatori, ci fanno capire che nel caso degli interpreti, il loro modo di operare rende praticamente impossibile l'interfacciamento con l'**Assembly**; del resto sappiamo che l'aspetto piu' negativo di questo tipo di programmi e' rappresentato dall'estrema lentezza di esecuzione che finisce per vanificare i benefici che si otterrebbero attraverso l'interfacciamento con l'**Assembly**.
Nel caso invece dei linguaggi compilati, l'interfacciamento con l'**Assembly** puo' essere effettuato in modo relativamente semplice; osserviamo infatti che anche in **Assembly** la generazione dell'eseguibile comporta una fase di assemblaggio che converte i moduli sorgente in moduli oggetto, e una fase di linking che collega tra loro i vari moduli oggetto producendo l'eseguibile finale. E' intuitivo quindi il fatto che per rendere possibile l'interfacciamento tra l'**Assembly** e i linguaggi compilati, sono necessarie due condizioni fondamentali:

a) Il formato interno dei moduli oggetto prodotti dall'assembler deve coincidere con il formato interno dei moduli oggetto prodotti dal compilatore.
b) I moduli scritti in **Assembly** devono attenersi a tutte le convenzioni seguite dal compilatore in relazione alla struttura a basso livello del programma.

La prima condizione viene rispettata attraverso la definizione di un formato standard per i moduli oggetto; nel mondo **DOS/Windows** i due formati standard piu' usati sono il formato **OMF** (Object Module Format), e il formato **COFF** (Common Object File Format). Come gia' sappiamo, il formato **OMF** viene supportato dal **MASM**, dal **TASM** e da molti compilatori della **Microsoft** e della **Borland**; il formato **COFF** invece viene largamente utilizzato in ambiente **Windows** dagli strumenti di sviluppo della **Microsoft** come **MASM**, **Visual C++**, etc. Si tenga presente che il formato **COFF** non viene supportato dagli strumenti di sviluppo della **Borland**; per maggiori dettagli su questi due formati per i moduli oggetto si consiglia di scaricare la documentazione ufficiale disponibile nella sezione **Downloads - Documentazione** di questo sito.
Per poter rispettare anche la seconda condizione, dobbiamo essere in grado di conoscere tutti i dettagli relativi alle convenzioni seguite dai compilatori per definire la struttura a basso livello di un programma eseguibile; attenendoci a queste convenzioni possiamo ottenere moduli **Assembly** interfacciabili con i moduli oggetto prodotti dai compilatori. Come e' stato detto nei capitoli dal 24 al 27, queste convenzioni riguardano in particolare, i modelli di memoria, i nomi e gli attributi dei segmenti di programma, svariati aspetti relativi alle procedure come il **prolog code**, l'**epilog code**, la creazione di variabili locali nello stack, i registri da utilizzare per i valori di ritorno, etc; uno degli scopi piu' importanti di queste convenzioni e' quello di ottenere la massima semplificazione possibile nella complessa fase di progettazione di un compilatore. In sostanza, al momento di convertire in codice macchina un programma scritto in **C/C++**, **Pascal**, **FORTRAN**, **BASIC**, etc, i compilatori si attengono ad una serie di convenzioni che possono essere considerate come veri e propri standard di linguaggio; in genere questi standard vengono decisi dai produttori di CPU (come la **Intel**), e da chi scrive i **SO** (come la **Microsoft**). Per un programmatore conoscere tutti questi aspetti significa poter interfacciare facilmente l'**Assembly** con i linguaggi di alto livello.
A partire dalla versione **5.x** il **MASM** offre il pieno supporto di queste convenzioni attraverso una serie di caratteristiche avanzate che dal punto di vista del programmatore si presentano come una vera e propria estensione del linguaggio **Assembly**; grazie all'uso di una sintassi evoluta, molto simile a quella dei linguaggi di alto livello, e' possibile ottenere notevoli semplificazioni nel lavoro

di interfacciamento tra l'**Assembly** e gli altri linguaggi.
Il compito fondamentale delle caratteristiche avanzate di **MASM** e' quello di consentire all'assembler la generazione automatica di tutte quelle porzioni di codice macchina che in base a quanto e' stato appena detto si possono considerare standard; in questo modo il programmatore ha la possibilita' di delegare all'assembler la gestione di tutti quegli aspetti che coinvolgono i modelli di memoria, i segmenti di programma, le convenzioni per le procedure, etc.
Naturalmente, e' anche possibile sfruttare queste caratteristiche avanzate per rendere piu' semplice lo sviluppo di programmi interamente scritti in **Assembly**; in questo caso pero' bisogna dire che come al solito, tutte queste "innovazioni" non suscitano particolare entusiasmo tra i puristi del linguaggio **Assembly**.
Gli utilizzatori del **TASM** che intendono servirsi di queste caratteristiche avanzate, devono munirsi della versione **5.x** o superiore dell'assembler della **Borland** (**TASM32.EXE**); le versioni precedenti del **TASM** offrono invece un supporto parziale delle nuove funzionalita' introdotte dal **MASM**. Si tenga presente comunque che in questo specifico settore, tra i due assembler esitono purtroppo numerose incompatibilita' spesso legate a banalissimi dettagli sintattici; la situazione e' aggravata dal fatto che la sintassi di queste estensioni dell'**Assembly** varia anche da una versione all'altra dello stesso assembler. Nei limiti del possibile, nel seguito del capitolo vengono indicate di volta in volta le eventuali differenze sintattiche esistenti tra **MASM** e **TASM**; per maggiori dettagli si consiglia di fare riferimento direttamente ai manuali dell'assembler che si intende utilizzare.

28.1 La direttiva .MODEL

Per poter accedere alle caratteristiche avanzate di **MASM** e **TASM** e' necessario servirsi della direttiva **.MODEL**; questa direttiva, che deve essere inserita all'inizio di ogni modulo **Assembly**, ci permette di passare all'assembler una serie di importanti informazioni relative in particolare al modello di memoria da utilizzare e alle convenzioni di linguaggio per le procedure. Tra tutte queste informazioni, l'unica obbligatoria e' quella relativa al modello di memoria; i principali modelli di memoria sono stati esaminati nel Capitolo 26. Per specificare un determinato modello di memoria, la sintassi da utilizzare e':

```
.MODEL memory_model
```

Il parametro **memory_model** e' uno tra quelli indicati dalla Figura 1; ogni modello di memoria indica il tipo di indirizzamento predefinito (**NEAR** o **FAR**) che l'assembler deve utilizzare per accedere alle procedure e ai dati del programma.

Figura 1 - Modelli di memoria

Parametro	**Indirizzamento**	
	Proc.	**Dati**
TINY	NEAR	NEAR
SMALL	NEAR	NEAR
MEDIUM	FAR	NEAR
COMPACT	NEAR	FAR
LARGE	FAR	FAR
HUGE	FAR	FAR
FLAT	NEAR	NEAR

Consideriamo la direttiva:

```
.MODEL TINY
```

Come gia' sappiamo, questo modello di memoria e' compatibile con la generazione di un eseguibile in formato **COM** (COre iMage); come abbiamo visto nel Capitolo 26, questo modello di memoria prevede la presenza di un unico segmento di programma contenente codice, dati e stack. In assenza di diverse indicazioni da parte del programmatore, tutti i dati e le procedure del programma vengono considerati di tipo **NEAR**.

Consideriamo la direttiva:

```
.MODEL SMALL
```

Questo modello di memoria prevede la presenza di un unico segmento di codice e di un unico segmento di dati (comprendente lo stack); in assenza di diverse indicazioni da parte del programmatore, tutti i dati e le procedure del programma vengono considerati di tipo **NEAR**.

Consideriamo la direttiva:

```
.MODEL MEDIUM
```

Questo modello di memoria prevede la presenza di due o piu' segmenti di codice e di un unico segmento di dati (comprendente lo stack); in assenza di diverse indicazioni da parte del programmatore, tutte le procedure del programma vengono considerate di tipo **FAR**, mentre tutti i dati vengono considerati di tipo **NEAR**.

Consideriamo la direttiva:

```
.MODEL COMPACT
```

Questo modello di memoria prevede la presenza di un unico segmento di codice e di due o piu' segmenti di dati; in assenza di diverse indicazioni da parte del programmatore, tutte le procedure del programma vengono considerate di tipo **NEAR**, mentre tutti i dati vengono considerati di tipo **FAR**.

Consideriamo infine la direttiva:

```
.MODEL LARGE
```

Questo modello di memoria prevede la presenza di due o piu' segmenti di codice e di due o piu' segmenti di dati; in assenza di diverse indicazioni da parte del programmatore, tutti i dati e le procedure del programma vengono considerati di tipo **FAR**.

Si ricorda che il modello **HUGE** e' del tutto simile al modello **LARGE**; l'unica differenza e' data dal fatto che nel modello **HUGE**, i compilatori (che supportano tale modello) utilizzano gli indirizzi logici normalizzati per permettere la gestione di grossi dati (come i vettori) che occupano piu' di **64** Kb in memoria.

Il modello **FLAT** viene supportato solo dalle versioni piu' recenti di **MASM** e **TASM**, e deve essere utilizzato per lo sviluppo di applicazioni destinate ai **SO** a **32** bit come **Windows 9x**, **Windows 2000**, **Windows XP**, **OS/2**, etc; questi sistemi operativi sono in grado di sfruttare l'architettura a **32** bit delle CPU **80386** e superiori. Sfruttare l'architettura a **32** bit di una CPU significa poter indirizzare in modo lineare (**flat**) la memoria del computer attraverso l'uso di offset a **32** bit; con un offset a **32** bit possiamo rappresentare tutti i valori compresi tra **00000000h** e **FFFFFFFFh**, con la possibilita' quindi di poter accedere in modo lineare a ben **4** Gb di **RAM**.
Nel caso generale, i **SO** a **32** bit supportano la possibilita' di far girare "contemporaneamente" piu' programmi (**multitasking**); per ciascun programma in esecuzione il **SO** crea uno spazio di indirizzamento "virtuale" da **4** Gb. Ad ogni programma in memoria viene fatto credere di essere l'unico in esecuzione, con tutto l'hardware del computer a sua completa disposizione; all'interno del suo spazio virtuale (**flat segment**) un programma assume una struttura del tutto simile a quella del modello di memoria **SMALL**. In sostanza, all'interno del segmento virtuale da **4** Gb viene creato un unico blocco codice, un unico blocco dati e un unico blocco stack; il blocco dati e il blocco stack vengono in genere raggruppati tra loro attraverso la direttiva **GROUP**. In una situazione di questo genere, per l'accesso alle procedure e ai dati del programma vengono utilizzati indirizzamenti di tipo **NEAR**; e' chiaro quindi che nel modello di memoria **FLAT** il termine **NEAR** indica un indirizzo logico formato dalla sola componente **offset** che questa volta pero' e' formata non da **16** bit ma da **32** bit.

Il modello di memoria **FLAT** dei **SO** a **32** bit rappresenta un vero e proprio paradiso per tutti quei programmatori che provengono dall'inferno della segmentazione a **64** Kb dei **SO** a **16** bit; per una descrizione piu' dettagliata del modello di memoria **FLAT** si puo' consultare la sezione **Win32 Assembly** di questo sito.

Se il programmatore vuole aggirare le convenzioni di Figura 1, deve passare all'assembler le opportune informazioni; nel caso ad esempio di un programma con modello di memoria **SMALL**, nessuno ci impedisce di definire procedure di tipo **FAR**. Queste procedure devono essere chiamate con una **FAR** call, e devono contenere di conseguenza un **FAR** return; per obbligare l'assembler a rispettare queste condizioni dobbiamo usare il **Model Modifier** (modificatore del modello di memoria) **FAR** e l'operatore **FAR PTR**. Supponiamo ad esempio di voler definire una procedura **FAR** chiamata **FarProc** in un programma con modello di memoria **SMALL**; in questo caso la definizione della procedura deve iniziare con:

```
FarProc proc far
```

In questo modo l'assembler capisce che l'istruzione **RET** posta alla fine della procedura deve essere convertita nel codice macchina di un **FAR** return.
Naturalmente, al momento di chiamare la procedura **FarProc** dobbiamo anche ricordarci di scrivere:

```
call far ptr FarProc
```

In questo modo l'assembler capisce che l'istruzione **CALL** deve essere convertita nel codice macchina di una **FAR** call; al momento di eseguire la chiamata di **FarProc**, la CPU inserira' quindi nello stack l'indirizzo di ritorno completo **seg:offset**, anche se il contenuto di **CS** rimane invariato.
Tra i parametri facoltativi che possiamo passare alla direttiva **.MODEL**, c'e' in particolare il parametro di linguaggio; attraverso questo parametro indichiamo all'assembler quale convenzione di linguaggio vogliamo utilizzare per la gestione delle procedure. Come gia' sappiamo, queste convenzioni riguardano: il **prolog code** (codice di ingresso), l'**epilog code** (codice di uscita), le variabili locali e la pulizia dello stack; specificando il parametro di linguaggio possiamo delegare tutti questi aspetti all'assembler. Per quanto riguarda la gestione del valore di ritorno, il programmatore deve attenersi alle convenzioni esposte nel Capitolo 24.
Per poter specificare un parametro di linguaggio la sintassi da utilizzare e':

```
.MODEL memory_model, language
```

Il parametro **language** e' uno tra quelli illustrati in Figura 2; per ogni parametro di linguaggio la Figura 2 indica l'ordine di inserimento degli argomenti di una procedura nello stack, e su chi ricade il compito di ripulire lo stack dagli argomenti stessi.

Figura 2 - Convenzioni di linguaggio

Parametro di Linguaggio	Ordine degli argomenti	Pulizia dello stack
BASIC	Sinistra-Destra	Procedura
FORTRAN	Sinistra-Destra	Procedura
PASCAL	Sinistra-Destra	Procedura
C	Destra-Sinistra	Caller
STDCALL	Destra-Sinistra	Procedura

Il **TASM** dispone anche dei parametri **CPP** e **PROLOG** per i linguaggi omonimi **C++** e **Prolog**; questi due linguaggi utilizzano le stesse convenzioni del linguaggio **C**.
In presenza del parametro di linguaggio l'assembler e' in grado di generare automaticamente il **prolog code** e l'**epilog code** delle procedure; il tutto avviene in conformita' alle convenzioni del linguaggio selezionato. Il programmatore gestisce tutta questa situazione attraverso l'uso di una sintassi evoluta, molto simile a quella dei linguaggi di alto livello; l'assembler elabora questa sintassi evoluta e la converte nell'opportuna sequenza di codici macchina.

Supponiamo ad esempio di voler creare un programma con modello di memoria **SMALL** conforme alle convenzioni **Pascal** per la gestione delle procedure; in questo caso dobbiamo scrivere:

```
.MODEL SMALL, PASCAL
```

Il parametro di linguaggio **PASCAL** dice all'assembler che in assenza di diverse indicazioni da parte del programmatore, ogni chiamata "evoluta" ad una procedura deve essere convertita in una sequenza di codici macchina che rispettano tutte le convenzioni del **Pascal**; in particolare, gli eventuali argomenti vengono automaticamente inseriti nello stack a partire dal primo (sinistra destra), mentre alla fine della procedura viene aggiunto all'istruzione **RET** un opportuno valore immediato da sommare a **SP** per la pulizia dello stack.

Il parametro **STDCALL** viene largamente utilizzato in ambiente **Windows** a **32** bit e rappresenta un misto tra le convenzioni **C** e quelle **Pascal**. Come gia' sappiamo, in **C** il passaggio degli argomenti ad una procedura avviene a partire dall'ultimo (destra sinistra), mentre la pulizia dello stack spetta al caller; questa convenzione permette di implementare procedure che accettano un numero variabile di argomenti. In **Pascal** invece il passaggio degli argomenti ad una procedura avviene a partire dal primo (sinistra destra), mentre la pulizia dello stack spetta alla procedura stessa; la pulizia dello stack in stile **Pascal** e' piu' veloce rispetto al caso del **C**. La convenzione **STDCALL** rappresenta un compromesso tra **C** e **Pascal** in quanto sfrutta la convenzione **C** per il passaggio degli argomenti, e la convenzione **Pascal** per la pulizia dello stack.

Se il programmatore vuole aggirare le convenzioni illustrate in Figura 2, deve passare le opportune informazioni all'assembler; in sostanza, se stiamo utilizzando ad esempio il parametro di linguaggio **PASCAL**, possiamo ugualmente definire procedure che seguono le convenzioni **C**. Per ottenere questo risultato dobbiamo inserire il **Language Modifier** (modificatore di linguaggio) **C** direttamente nella definizione di quelle procedure che vogliono utilizzare le convenzioni **C**. La sintassi da utilizzare viene illustrata piu' avanti.

28.2 Sintassi avanzata per le procedure

Analizziamo ora un esempio pratico che ci permette di illustrare la sintassi avanzata di **MASM** e **TASM** per la gestione delle procedure; a tale proposito, supponiamo di voler creare un programma con modello di memoria **LARGE**, conforme alle convenzioni **C** per le procedure. Prima di tutto quindi dobbiamo scrivere:

```
.MODEL LARGE, C
```

A questo punto procediamo con l'implementazione di una procedura che presenta il seguente prototipo **C**:

```
int far TestProc(long param1, int param2, int param3);
```

Questa procedura quindi ha tre parametri che sono: **param1** (intero con segno a **32** bit), **param2** (intero con segno a **16** bit) e **param3** (intero con segno a **16** bit); la procedura inoltre termina restituendo in **AX** un intero con segno a **16** bit. Supponiamo anche che **TestProc** abbia bisogno di due variabili locali chiamate **locVar1** (intero con segno a **32** bit) e **locVar2** (intero con segno a **16** bit); per poter chiamare questa procedura definiamo le quattro variabili visibili in Figura 3.

Figura 3 - Segmento dati del programma

```
varDword    dd    000D3BA2h
varWord1    dw    00F2h
varWord2    dw    003Dh
retVal      dw    0000h
```

In linguaggio **C** la chiamata di **TestProc** con i primi tre argomenti visibili in Figura 3 assume il seguente aspetto:
```
retVal = TestProc(varDword, varWord1, varWord2);
```
Come si puo' notare, il valore di ritorno di **TestProc** viene salvato in **retVal**; naturalmente, il compilatore **C** provvede a trasferire in **retVal** il valore a **16** bit che **TestProc** ha inserito in **AX**.
Traducendo questa chiamata in **Assembly** classico otteniamo:

```
push     varWord2       ; passa varWord2
push     varWord1       ; passa varWord1
push     varDword       ; passa varDword
call     TestProc       ; chiama TestProc
add      sp, 8          ; pulisce lo stack
mov      retVal, ax     ; salva il valore di ritorno
```

Nel rispetto delle convenzioni **C**, gli argomenti vengono passati alla procedura a partire dall'ultimo; l'ordine da seguire e' rappresentato quindi dai **2** byte di **varWord2** seguiti dai **2** byte di **varWord1** e dai **4** byte di **varDword**. Quando il controllo torna al caller, viene effettuata la pulizia dello stack; siccome abbiamo inserito **8** byte nello stack, la pulizia consiste nel sommare il valore immediato **8** a **SP**. La procedura **TestProc** restituisce il suo valore di ritorno in **AX**; in **Assembly** il compito di leggere questo valore e memorizzarlo in **retVal** spetta ovviamente al programmatore.
Per analizzare in dettaglio quello che accade nello stack, partiamo dalla condizione **SP=0202h** e supponiamo che l'indirizzo di ritorno sia **seg:offset = 0CF2h:003Bh**; la chiamata **FAR** di **TestProc** comporta i seguenti passi eseguiti dalla CPU:

***** la CPU sottrae due byte a **SP** e ottiene **SP=0200h**;
***** il contenuto **003Dh** di **varWord2** viene inserito a **SP=0200h**;
***** la CPU sottrae **2** byte a **SP** e ottiene **SP=01FEh**;
***** il contenuto **00F2h** di **varWord1** viene inserito a **SP=01FEh**;
***** la CPU sottrae **2** byte a **SP** e ottiene **SP=01FCh**;
***** la word piu' significativa **000Dh** di **varDword** viene inserita a **SP=01FCh**;
***** la CPU sottrae **2** byte a **SP** e ottiene **SP=01FAh**;
***** la word meno significativa **3BA2h** di **varDword** viene inserita a **SP=01FAh**;
***** la CPU sottrae **2** byte a **SP** e ottiene **SP=01F8h**;
***** la componente **seg=0CF2h** dell'indirizzo di ritorno viene inserita a **SP=01F8h**;
***** la CPU sottrae **2** byte a **SP** e ottiene **SP=01F6h**;
***** la componente **offset=003Bh** dell'indirizzo di ritorno viene inserita a **SP=01F6h**;
***** l'indirizzo **seg:offset** di **TestProc** viene caricato in **CS:IP** e viene effettuato un salto **FAR** a **CS:IP**.

Il salto **FAR** ci porta all'interno di **TestProc**; la Figura 4 mostra l'aspetto che assume questa procedura in stile **Assembly** classico:

Figura 4 - Procedura TestProc

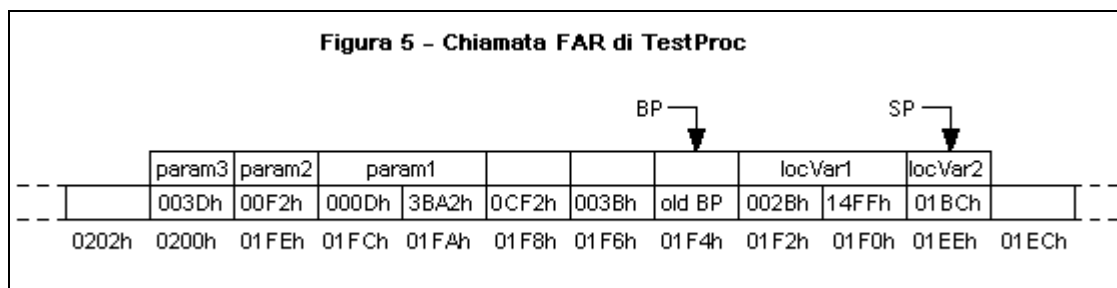
```
TestProc proc far

   param1      equ   [bp+6]    ; parametro param1
   param2      equ   [bp+10]   ; parametro param2
   param3      equ   [bp+12]   ; parametro param3
   locVar1     equ   [bp-4]    ; var. locale locVar1
   locVar2     equ   [bp-6]    ; var. locale locVar2

   push        bp              ; preserva bp
   mov         bp, sp          ; ss:bp = ss:sp
   sub         sp, 6           ; spazio per le var. locali

   ....................        ; corpo di TestProc

   mov         ax, ....        ; ax = return value

   mov         sp, bp          ; pulizia var. locali
   pop         bp              ; ripristina bp
   ret                         ; far return

TestProc endp
```

Per giustificare questa situazione ricordiamoci innanzi tutto che appena entrati in **TestProc** abbiamo **SP=01F6h**; le prime tre istruzioni presenti all'interno di **TestProc** sono le seguenti:

```
push     bp       ; preserva bp
mov      bp, sp   ; ss:bp = ss:sp
sub      sp, 6    ; spazio per le var. locali
```

Come gia' sappiamo, queste tre istruzioni ci servono per accedere ai parametri e alle variabili locali di **TestProc**; quando la CPU incontra queste tre istruzioni esegue le seguenti operazioni:

***** la CPU sottrae due byte a **SP** e ottiene **SP=01F4h**;
***** il contenuto (che chiamiamo simbolicamente **old BP**) di **BP** viene inserito a **SP=01F4h**;
***** il contenuto **01F4h** di **SP** viene copiato in **BP** per cui si ottiene **BP=01F4h**;
***** la CPU sottrae **6** byte a **SP** e ottiene **SP=01EEh**.

Quest'ultima operazione serve per ottenere **6** byte di spazio nello stack, da destinare alle variabili locali; come e' stato detto in precedenza, **TestProc** ha bisogno di due variabili locali chiamate simbolicamente **locVar1** (**4** byte) e **locVar2** (**2** byte).
Tutte le operazioni che comprendono l'inserimento degli argomenti nello stack e le tre precedenti istruzioni, rappresentano il **prolog code** di **TestProc**; al termine del **prolog code** otteniamo quindi **BP=01F4h** e **SP=01EEh**. Per maggiore chiarezza, tutta questa situazione viene illustrata in Figura 5.

L'aspetto importante da ricordare e' dato dal fatto che con questo sistema, i parametri di **TestProc** vengono a trovarsi a spiazzamenti positivi rispetto a **BP**, mentre le variabili locali vengono a trovarsi a spiazzamenti negativi rispetto a **BP**; dalla Figura 5 infatti ricaviamo le seguenti informazioni:

Il parametro **param1** che e' la copia di **varDword**, viene a trovarsi nello stack all'offset:
`BP+6 = 01F4h + 0006h = 01FAh`.
Il parametro **param2** che e' la copia di **varWord1**, viene a trovarsi nello stack all'offset:
`BP+10 = 01F4h + 000Ah = 01FEh`.
Il parametro **param3** che e' la copia di **varWord2**, viene a trovarsi nello stack all'offset:
`BP+12 = 01F4h + 000Ch = 0200h`.
La variabile locale **locVar1** viene a trovarsi nello stack all'offset:
`BP-4 = 01F4h - 0004h = 01F0h`.
La variabile locale **locVar2** viene a trovarsi nello stack all'offset:
`BP-6 = 01F4h - 0006h = 01EEh`.

E' importante osservare che **param1** si trova all'offset **BP+6** e occupa **4** byte nello stack; di conseguenza **param2** si trovera' **4** byte piu' avanti, e cioe' all'offset **BP+10**. Analogamente, la variabile locale **locVar1** e' formata da **4** byte, e quindi si trova all'offset **BP-4**; la variabile locale **locVar2** occupa invece **2** byte e si trova quindi all'offset **BP-6**. Si tenga presente che in Figura 5, i valori **002B14FFh** per **locVar1**, e **01BCh** per **locVar2** sono solo dei valori di esempio; in assenza di inizializzazione, tutte le variabili locali di un programma hanno un contenuto casuale dovuto al fatto che lo stack viene continuamente scritto e sovrascritto.
Qualcuno potrebbe avere dei dubbi in relazione agli offset delle due variabili locali **locVar1** e **locVar2**; per chiarire questi dubbi bisogna osservare innanzi tutto che la porzione di memoria visibile in Figura 5 viene disposta come al solito con gli indirizzi crescenti da destra verso sinistra. Ricordiamoci inoltre che l'offset di una locazione di memoria coincide con l'offset del suo byte meno significativo; di conseguenza, la variabile locale **locVar1** parte dall'offset **BP-4** ed occupa nello stack i **4** byte che si trovano agli offset **BP-4**, **BP-3**, **BP-2** e **BP-1**. Lo stesso discorso vale anche per **locVar2**; questa variabile locale infatti parte dall'offset **BP-6** ed occupa nello stack i **2** byte che si trovano agli offset **BP-6** e **BP-5**.
Le considerazioni appena svolte dimostrano ancora una volta quanto sia importante in questi casi tracciarsi su un foglio di carta uno schema dello stack come quello visibile in Figura 5; se si prova a svolgere mentalmente tutti questi calcoli si va incontro sicuramente a numerosi errori. Per gestire facilmente i parametri e le variabili locali di **TestProc** conviene come al solito dichiarare le macro alfanumeriche visibili in Figura 4.

Passiamo ora alla descrizione della fase di terminazione di **TestProc** con conseguente restituzione del controllo al caller. Prima di tutto all'interno di **TestProc** dobbiamo caricare in **AX** il **return value** della procedura; nel nostro caso utilizziamo **AX** in quanto dobbiamo restituire un intero a **16** bit.

A questo punto comincia l'**epilog code** di **TestProc**; il primo compito da svolgere consiste nel rimuovere dallo stack le variabili locali. Questo compito che spetta sempre alla procedura (indipendentemente dalle convenzioni di linguaggio), viene svolto dall'istruzione:

```
mov sp, bp
```

In questo modo si ottiene **SP=01F4h**; si puo' anche scrivere:

```
add sp, 6
```

Questo secondo metodo pero' ci espone a dei potenziali errori in quanto ci obbliga a ricordare esattamente quanti byte di variabili locali avevamo richiesto allo stack; naturalmente il primo metodo funziona solo se **BP** non ha subito nessuna modifica imprevista.
Il secondo compito da svolgere consiste nel ripristinare **BP** ponendo **BP=oldBP**; questo lavoro viene svolto dall'istruzione:

```
pop bp
```

Subito dopo l'estazione di **old BP** dallo stack, la CPU somma **2** byte a **SP** e ottiene **SP=01F6h**; a questo punto viene incontrato un **FAR** return che comporta le seguenti operazioni svolte dalla CPU:

***** la CPU estrae dall'offset **SP=01F6h** la word **003Bh** e la carica in **IP**;
***** la CPU somma **2** byte a **SP** e ottiene **SP=01F8h**;
***** la CPU estrae dall'offset **SP=01F8h** la word **0CF2h** e la carica in **CS**;
***** la CPU somma **2** byte a **SP** e ottiene **SP=01FAh**;
***** la CPU salta a **CS:IP = 0CF2h:003Bh**.

Subito dopo il salto **FAR** ci ritroviamo nell'istruzione associata all'indirizzo di ritorno; come si vede nella chiamata di **TestProc**, questa istruzione e':

```
add sp, 8
```

In questo modo stiamo restituendo allo stack gli **8** byte utilizzati per il passaggio degli argomenti a **TestProc**; come sappiamo infatti, in **C** questo compito spetta al caller.
Subito dopo questa istruzione otteniamo **SP=0202h** e in questo modo abbiamo ripristinato il valore che **SP** aveva prima della chiamata di **TestProc**; al termine di tutte queste fasi il registro **AX** contiene il valore di ritorno della procedura.

Le considerazioni appena svolte si riferiscono al caso in cui il "rude" programmatore **Assembly** si assuma l'onore e l'onere di gestire in proprio il **prolog code** e l'**epilog code** delle procedure; come si puo' constatare, questi concetti una volta appresi risultano abbastanza semplici da applicare.
Vediamo ora come si modifica la situazione nel momento in cui decidiamo di servirci della sintassi avanzata di **MASM** e **TASM**; come e' stato detto in precedenza, per poter usufruire di queste innovazioni dobbiamo utilizzare la direttiva **.MODEL**. Nel nostro caso il programma inizia con:

```
.MODEL LARGE, C
```

La prima novita' che la sintassi avanzata di **MASM** e **TASM** ci mette a disposizione, e' rappresentata dalla possibilita' di dichiarare anche in **Assembly** un prototipo per ogni procedura del programma che stiamo scrivendo; come accade nei linguaggi di alto livello, i prototipi permettono all'assembler di verificare la correttezza delle chiamate alle procedure, generando se necessario gli opportuni messaggi di errore. In sostanza, incontrando una chiamata ad una procedura l'assembler verifica se il numero e il tipo degli argomenti che stiamo passando coincide con le informazioni presenti nel prototipo; la struttura generale di un prototipo di procedura assume in **Assembly** il seguente aspetto:

```
nome_procedura PROTO [model_modifier] [language_modifier] [argument_list]
```

Le parentesi quadre indicano che questi parametri sono tutti facoltativi; con le vecchie versioni di **TASM**, al posto di **PROTO** e' necessario utilizzare **PROCDESC**.
Il parametro **argument_list** (lista argomenti) elenca il tipo di ogni argomento da passare alla procedura; ciascun tipo e' preceduto dal simbolo **:** (due punti) e i vari tipi sono separati tra loro da virgole. I nomi simbolici dei vari argomenti (**parametri formali**) sono facoltativi; come accade nel linguaggio **C**, usualmente questi nomi vengono omessi. L'ordine con il quale gli argomenti vengono

disposti nel prototipo non ha niente a che vedere con le convenzioni per il passaggio degli argomenti alle procedure; indipendentemente quindi dal parametro di linguaggio utilizzato, gli argomenti di **TestProc** devono essere disposti sempre nello stesso ordine.
Il modificatore di modello di memoria deve essere inserito solo se vogliamo aggirare l'analogo parametro specificato nella direttiva **.MODEL**; lo stesso discorso vale per il modificatore di linguaggio. Il luogo ideale per l'inserimento dei prototipi delle procedure e' la parte iniziale del modulo **Assembly**; naturalmente, il prototipo di una procedura deve precedere la chiamata della procedura stessa.
Per applicare le cose appena dette a **TestProc** dobbiamo fare i conti con le differenze sintattiche esistenti tra **MASM** e **TASM**; nel caso del **MASM** dobbiamo scrivere:

```
TestProc PROTO FAR C :DWORD, :WORD, :WORD
```

Nel caso del **TASM** dobbiamo invece scrivere:

```
TestProc PROTO C FAR :DWORD, :WORD, :WORD
```

Osserviamo che i modificatori di linguaggio e di modello di memoria coincidono con quelli presenti nella direttiva **.MODEL**, e quindi sono superflui; il prototipo di **TestProc** puo' essere riscritto quindi per entrambi gli assembler come:

```
TestProc PROTO :DWORD, :WORD, :WORD
```

Se vogliamo essere pignoli possiamo anche specificare i nomi simbolici dei parametri di **TestProc**; in questo caso dobbiamo scrivere:

```
TestProc PROTO param1 :DWORD, param2 :WORD, param3 :WORD
```

E' importante tener presente che i prototipi delle procedure svolgono anche il ruolo della direttiva **EXTRN**; in pratica, se **TestProc** e' definita in un modulo diverso da quello del caller, la presenza del prototipo rende inutile la direttiva **EXTRN**.

Una volta dichiarato il prototipo, possiamo passare alla definizione della procedura; nel caso generale la definizione di una procedura inizia con la seguente intestazione in versione **MASM**:

```
nome_procedura PROC [model_modifier] [language_modifier] [argument_list]
```

Il **MASM** richiede che l'intestazione di una procedura sia identica al prototipo; di conseguenza, gli eventuali parametri **model_modifier** e **language_modifier** specificati nel prototipo, devono essere specificati nello stesso ordine anche nell'intestazione della procedura.
Con il **TASM** invece bisogna scrivere:

```
nome_procedura PROC [argument_list]
```

Il **TASM** si serve infatti del prototipo per conoscere tutti i dettagli relativi alla procedura; non e' necessario quindi specificare nell'intestazione della procedura gli eventuali parametri **model_modifier** e **language_modifier** gia' specificati nel prototipo della procedura stessa.
In generale possiamo notare che l'intestazione della procedura e' del tutto simile al suo prototipo; questa volta la lista **argument_list** deve specificare anche i nomi simbolici che intendiamo utilizzare per i parametri della procedura. I parametri **model_modifier** e **language_modifier** devono essere specificati solo se vogliamo aggirare gli analoghi parametri passati alla direttiva **.MODEL**.
All'interno della procedura possiamo definire le necessarie variabili locali attraverso la direttiva **LOCAL**; la sintassi generale da utilizzare e' la seguente:

```
LOCAL nome_var1 :tipo_var1, nome_var2 :tipo_var2, ...
```

Sia il **prolog_code** che l'**epilog_code** non vengono piu' gestiti dal programmatore; tutto questo lavoro infatti viene delegato all'assembler.

Applichiamo ora tutti questi concetti al caso di **TestProc**; la Figura 6 illustra il nuovo aspetto assunto dalla procedura **TestProc** con l'uso della sintassi avanzata di **MASM** e **TASM**.

Figura 6 - Sintassi avanzata per TestProc

```
TestProc proc param1 :DWORD, param2: WORD, param3 :WORD

   LOCAL       locVar1 :DWORD, locVar2: WORD

   ....................       ; corpo di TestProc

   mov         ax, ....       ; ax = return value

   ret                        ; far return

TestProc endp
```

La prima cosa da osservare riguarda il fatto che questa volta, come accade nei linguaggi di alto livello, la definizione di **TestProc** deve specificare anche la lista dei parametri che la procedura richiede; e' chiaro che il numero e il tipo dei parametri di questa lista deve coincidere esattamente con le analoghe informazioni specificate nel prototipo di **TestProc**.
Subito dopo l'intestazione di **TestProc** incontriamo la direttiva **LOCAL**; come si puo' notare, questa direttiva ci permette di utilizzare una sintassi molto sofisticata per la definizione delle variabili locali. Come abbiamo visto nel Capitolo 23, la direttiva **LOCAL** viene anche utilizzata per dichiarare etichette con visibilita' locale all'interno del corpo di una macro alfanumerica.
In Figura 6 notiamo che dopo la direttiva **LOCAL** inizia subito il corpo della procedura; inoltre, al termine di **TestProc** troviamo solamente il caricamento in **AX** del valore di ritorno e l'istruzione **RET**.
Come e' stato detto in precedenza, il **prolog code** e l'**epilog code** devono essere omessi in quanto vengono gestiti direttamente dall'assembler; per poter delegare questo lavoro all'assembler dobbiamo necessariamente chiamare le procedure attraverso la sintassi avanzata di **MASM** e **TASM**. In pratica, dobbiamo servirci di due nuove potentissime direttive chiamate **CALL** per il **TASM** e **INVOKE** per il **MASM**; analizziamo quindi le caratteristiche di queste direttive.

28.3 Le direttive CALL (TASM) e INVOKE (MASM)

Con le caratteristiche avanzate di **MASM** e **TASM**, le procedure possono essere chiamate attraverso l'uso di una sintassi molto simile a quella dei linguaggi di alto livello; a tale proposito dobbiamo servirci di una apposita direttiva che purtroppo varia da assembler ad assembler. Nel caso del **MASM** questa direttiva viene chiamata **INVOKE**; nel caso del **TASM** invece questa direttiva viene chiamata **CALL**, e non deve essere confusa con l'omonima istruzione della CPU. La sintassi generale da utilizzare con il **MASM** e':

```
INVOKE nome_procedura, [argument_list]
```

La sintassi generale da utilizzare con il **TASM** e':

```
CALL nome_procedura, [argument_list]
```

La **argument_list** rappresenta ovviamente la lista dei nomi degli argomenti (**parametri attuali**) da passare alla procedura; i vari nomi sono separati da virgole. E' importante notare anche la presenza della virgola dopo il **nome_procedura**.

Applicando questi concetti alla procedura **TestProc**, nel caso del **MASM** otteniamo:

```
invoke   TestProc, varDword, varWord1, varWord2
mov      retVal, ax
```

Nel caso del **TASM** invece otteniamo:

```
call      TestProc, varDword, varWord1, varWord2
mov       retVal, ax
```

Come si puo' notare, il programmatore deve preoccuparsi esclusivamente della chiamata della procedura e del salvataggio dell'eventuale valore di ritorno; questo salvataggio deve essere effettuato subito dopo la chiamata della procedura per evitare che il contenuto di **AX** venga inavvertitamente sovrascritto.
Tutto il lavoro legato al **prolog_code** e all'**epilog_code** viene svolto dall'assembler che incontrando questo tipo di chiamata, produce esattamente tutto il codice che abbiamo analizzato in precedenza (nel caso della gestione in stile classico di **TestProc**); per renderci conto del lavoro svolto dall'assembler possiamo richiedere come al solito il **listing file** del modulo che vogliamo assemblare. In questo modo possiamo constatare che in presenza della precedente chiamata a **TestProc**, e del parametro di linguaggio **C**, l'assembler crea il codice macchina necessario per l'inserimento degli argomenti nello stack a partire da quello piu' a destra, e per la pulizia dello stack da parte del caller; inoltre, in presenza del modello di memoria **LARGE**, l'assembler crea il codice macchina per una **FAR** call a **TestProc** e per un conseguente **FAR** return. Tutti questi dettagli vengono evidenziati dal **listing file** generato dal **TASM**, mentre nel caso del **MASM** stranamente questo non accade; evidentemente la **Microsoft** vuole tenere nascosto il tipo di codice macchina prodotto dalla direttiva **INVOKE**.

Sicuramente l'aspetto piu' potente delle caratteristiche avanzate di **MASM** e **TASM** e' rappresentato dal fatto che per adattare un intero modulo **Assembly** ad un diverso modello di memoria e/o ad una diversa convenzione di linguaggio, ci basta modificare solamente gli opportuni parametri passati alla direttiva **.MODEL**; nel nostro caso possiamo scrivere ad esempio:

```
.MODEL SMALL, PASCAL
```

In un caso del genere l'assembler converte automaticamente tutte le procedure da **FAR** a **NEAR**, e utilizza le convenzioni del linguaggio **Pascal** per la gestione delle procedure stesse; inoltre gli indirizzi **NEAR** vengono utilizzati anche per l'accesso ai dati del programma.

Per quanto riguarda l'incompatibilita' tra le direttive **CALL** e **INVOKE**, si puo' risolvere in parte il problema attraverso una macro alfanumerica; se ad esempio vogliamo assemblare con **TASM** un programma scritto con **MASM** e contenente quindi numerose direttive **INVOKE**, possiamo definire la seguente macro:

```
INVOKE equ CALL
```

In questo modo l'assembler ogni volta che incontra la macro **INVOKE**, la converte nella stringa **CALL**.
Come e' stato appena detto, questa macro offre una soluzione parziale del problema in quanto la direttiva **INVOKE** presenta alcune caratteristiche del tutto incompatibili con la direttiva **CALL**; questo aspetto viene analizzato piu' avanti.

Un'altra considerazione importante riguarda il fatto che la direttiva **INVOKE** per svolgere il suo lavoro potrebbe eventualmente utilizzare i registri **AL**, **AH**, **AX**, **EAX**, **DL**, **DH**, **DX**, **EDX**; se utilizziamo questi registri nella **argument_list** di **INVOKE**, il loro contenuto potrebbe anche essere sovrascritto. In una situazione del genere il **MASM** visualizza il messaggio di errore:

```
register value overwritten by INVOKE
```

Con questo messaggio il **MASM** ci sta chiedendo di utilizzare altri registri al posto di quelli elencati in precedenza.

28.4 L'operatore ADDR di MASM

Nel Capitolo 24 dedicato ai sottoprogrammi, e' stata messa in evidenza la delicata situazione che si viene a creare quando all'interno di una procedura dobbiamo gestire l'offset di una variabile locale; in particolare abbiamo visto che in un caso del genere non avrebbe senso utilizzare l'operatore **OFFSET**. Questo operatore infatti lavora solo in fase di assemblaggio, e quindi richiede un operando che deve essere un identificatore definito staticamente nel programma; come sappiamo, questi identificatori hanno un offset ben preciso che viene assegnato dall'assembler (o dal linker nel caso di rilocazione) e rimane fisso per tutta la fase di esecuzione di un programma.
La situazione cambia radicalmente nel caso di una variabile locale; le variabili locali di una procedura, vengono create nello stack nel preciso momento in cui la procedura stessa viene chiamata dal caller. Questo significa che se chiamiamo **10** volte una stessa procedura, puo' capitare che ad ogni chiamata le sue variabili locali vengano create sempre ad offset differenti rispetto alla chiamata precedente; tutto dipende infatti dalla posizione in cui si trova **SP** nello stack al momento della chiamata. Bisogna anche ricordare che le variabili locali vengono distrutte al termine della procedura che le ha create; questo significa che una variabile locale esiste (ed e' quindi visibile) solo all'interno della procedura a cui appartiene.
Se ad esempio, all'interno di **TestProc** (vedi Figura 4) vogliamo caricare in **AX** l'offset di **locVar1**, non possiamo scrivere:

```
mov ax, offset locVar1
```

Questa istruzione carica infatti in **AX** un valore privo di senso; per aggirare il problema possiamo utilizzare l'istruzione **LEA** scrivendo:

```
lea ax, locVar1
```

L'assembler espande la macro **locVar1** e ottiene l'istruzione:

```
lea ax, [bp-4]
```

Come e' stato spiegato nel Capitolo 15, bisogna stare molto attenti al fatto che l'istruzione **LEA** potrebbe creare una certa confusione al programmatore; l'istruzione precedente carica in **AX** l'offset rappresentato da **BP-4** e non il contenuto a **16** bit della locazione **BP-4**.

Il problema appena esposto si presenta anche quando utilizziamo le direttive **CALL** o **INVOKE** all'interno di una procedura, e nella **argument_list** di queste direttive e' presente anche l'offset di qualche variabile locale; supponiamo ad esempio di dover chiamare una procedura **Proc2** dall'interno di **TestProc**. Se dobbiamo passare a **Proc2** il parametro **param2** e l'offset di **locVar2**, dovremmo scrivere:

```
call Proc2, param2, offset locVar2
```

Questa istruzione produce un risultato privo di senso che in genere manda in crash il programma; per risolvere il problema possiamo scrivere:

```
lea      ax, locVar2
call     Proc2, param2, ax
```

La direttiva **INVOKE** di **MASM** permette di rendere piu' compatto questo codice grazie all'operatore **ADDR**; l'operatore **ADDR** infatti equivale all'operatore **OFFSET** per le variabili statiche, e all'istruzione **LEA** per le variabili locali. Nel caso di **MASM** possiamo quindi scrivere:

```
invoke Proc2, param2, addr locVar2
```

L'operatore **ADDR** non viene supportato dal **TASM** e non puo' essere quindi utilizzato con la direttiva **CALL**; tutto cio' puo' far pensare che la direttiva **INVOKE** sia molto piu' sofisticata della direttiva **CALL** di **TASM**. Ricordiamoci pero' che **INVOKE** per poter svolgere il suo lavoro potrebbe sovrascrivere i registri citati in precedenza; la direttiva **CALL** di **TASM** invece puo' essere utilizzata con assoluta tranquillita' in quanto non presenta nessun inconveniente di questo genere.

28.5 Convenzioni per i segmenti di programma

Le convenzioni utilizzate dai compilatori coinvolgono anche i nomi e gli attributi dei segmenti di programma; in questo caso lo scopo principale di queste convenzioni e' quello di ottimizzare al massimo alcuni aspetti dei programmi come la velocita', la compattezza, le esigenze di memoria, etc.
Per organizzare un programma nel modo piu' efficiente possibile, i compilatori suddividono il codice e i dati in tante categorie; in particolare, i dati del programma vengono suddivisi in **dati costanti**, **dati non inizializzati** (compreso lo stack) e **dati inizializzati**. Per ciascuna categoria di codice o dati viene creato un apposito segmento di programma; in tutti i modelli di memoria, alcuni di questi segmenti di dati vengono sempre inseriti attraverso la direttiva **GROUP** in un gruppo chiamato convenzionalmente **DGROUP** (Data Group). I compilatori inizializzano poi **DS** ponendo **DS=DGROUP**; in questo modo possono accedere ai dati contenuti in **DGROUP** attraverso indirizzi di tipo **NEAR**. Possiamo dire quindi che il gruppo **DGROUP** rappresenta il blocco dati principale di un programma che segue queste convenzioni.

Analizziamo ora i nomi e gli attributi che per convenzione i compilatori assegnano ai vari segmenti di programma; nel seguito quando si parla di assembler o di assemblaggio, ci si riferisce in generale alla fase di traduzione del codice sorgente in codice macchina.

Per i dati inizializzati viene creato un segmento di programma avente le seguenti caratteristiche:
```
_DATA SEGMENT WORD PUBLIC USE16 'DATA'
```
All'interno di questo blocco vengono inserite tutte le variabili statiche che vengono inizializzate al momento stesso della loro definizione; nei linguaggi come il **C** o il **Pascal** queste variabili sono quelle che vengono definite e inizializzate al di fuori di qualsiasi funzione o procedura. In **C** possiamo avere ad esempio:
```
int var_word1 = 12500;
```
In **Pascal** possiamo avere ad esempio:
```
CONST var_word1: Integer = 12500;
```
In **Assembly** possiamo avere ad esempio:
```
var_word1 dw 12500
```
Le variabili definite in questo modo esistono per tutta la fase di esecuzione del programma; in alcuni linguaggi inoltre, queste variabili possono essere dichiarate **PUBLIC** in modo da renderle visibili anche in altri moduli.
Siccome il blocco **_DATA** e' dotato di nome e attributi standard, per poterlo definire possiamo servirci della apposita direttiva semplificata:
```
.DATA
```
Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.DATA** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.
In tutti i modelli di memoria il blocco **_DATA** viene sempre inserito nel gruppo **DGROUP**; si ricorda che i compilatori inizializzano **DS** ponendo **DS=DGROUP**; in questo modo possono accedere ai dati contenuti in **DGROUP** attraverso indirizzi di tipo **NEAR**.

Per i dati non inizializzati viene creato un segmento di programma avente le seguenti caratteristiche:
```
_BSS SEGMENT WORD PUBLIC USE16 'BSS'
```
All'interno di questo blocco vengono inserite tutte le variabili statiche che vengono definite ma non inizializzate; nei linguaggi come il **C** o il **Pascal** queste variabili sono quelle che vengono definite al di fuori di qualsiasi funzione o procedura. In **C** possiamo avere ad esempio:
```
int var_temp;
```
In **Pascal** possiamo avere ad esempio:
```
VAR var_temp: Integer;
```
In **Assembly** possiamo avere ad esempio:

```
var_temp dw ?
```

Le variabili definite in questo modo esistono per tutta la fase di esecuzione del programma; in alcuni linguaggi inoltre, queste variabili possono essere dichiarate **PUBLIC** in modo da renderle visibili anche in altri moduli.
Siccome il blocco **_BSS** e' dotato di nome e attributi standard, per poterlo definire possiamo servirci della apposita direttiva semplificata:

```
.DATA?
```

Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.DATA?** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.
Il blocco **_BSS** viene sempre inserito nel gruppo **DGROUP** insieme al blocco **_DATA**; per il blocco **_BSS** valgono quindi tutte le considerazioni svolte in precedenza per il blocco **_DATA**.

Per i dati costanti viene creato un segmento di programma avente le seguenti caratteristiche:

```
_CONST SEGMENT WORD PUBLIC USE16 'CONST'
```

All'interno di questo blocco vengono inserite tutte quelle costanti di tipo stringa o di tipo numerico che non appartengono necessariamente ad una variabile; in **C** possiamo avere ad esempio:

```
printf("Premere un tasto per continuare");
```

La stringa **"Premere un tasto per continuare"** viene usata come argomento da passare a **printf**, ma non e' associata a nessuna variabile di tipo stringa; questo e' un classico esempio di dato costante che viene inserito nel blocco **_CONST**.
In **Pascal** possiamo avere l'analogo esempio:

```
WriteLn('Premere un tasto per continuare');
```

Anche in **Assembly** possiamo utilizzare il blocco **_CONST** per definire dati costanti da assegnare poi di volta in volta a diverse variabili; nel blocco **_CONST** possiamo inserire ad esempio la seguente definizione:

```
str_const db 'Premere un tasto per continuare', 0
```

Nel blocco **_DATA** possiamo poi inserire la seguente definizione:

```
str_addr dw offset str_const
```

In questo modo la variabile **str_addr** a **16** bit viene inizializzata con l'offset di **str_const**; in seguito possiamo anche far puntare **str_addr** da un'altra parte, e possiamo assegnare l'offset di **str_const** ad un'altra variabile.
Tutti i dati inseriti nel blocco **_CONST** possono essere considerati di fatto come veri e propri dati statici che quindi esistono per tutta la fase di esecuzione del programma; una conseguenza importante di tutto cio' e' rappresentata dalla seguente definizione **C**:

```
char *pt_str = "Inserire un numero";
```

Quando il compilatore **C** incontra questa definizione, riserva **19** byte di memoria destinati a contenere i **18** caratteri della stringa piu' lo zero finale (stringa **C**); successivamente il compilatore richiede **2** byte di memoria e li assegna alla variabile puntatore **pt_str**. Nei **2** byte assegnati a **pt_str** viene caricato l'offset iniziale della stringa; a questo punto il compilatore ha creato una variabile **pt_str** che e' un puntatore **NEAR** alla stringa **"Inserire un numero"**. Anche se questa definizione viene inserita all'interno di una funzione, il compilatore crea **pt_str** nello stack, ma posiziona la stringa nel blocco **_CONST**; questo comportamento e' legato al fatto che le stringhe occupano troppo spazio nello stack. Possiamo dire quindi che **pt_str** e' una variabile locale, mentre la stringa e' un dato statico che non viene distrutto al termine della funzione. Tutto cio' significa che in **C** e' perfettamente legale restituire al caller l'indirizzo iniziale della stringa assegnata a **pt_str**; il caller infatti riceve l'indirizzo di un dato statico che esiste per tutta la fase di esecuzione del programma.
E' un errore invece restituire al caller l'indirizzo di **pt_str**; infatti **pt_str** e' una variabile locale che quindi viene distrutta al termine della funzione che l'ha creata.
Per quanto riguarda i dati costanti di tipo numerico, bisogna dire che a seconda della loro dimensione in byte, i compilatori possono decidere se inserirli nel blocco **_CONST** o direttamente nel codice macchina; consideriamo ad esempio la seguente istruzione **C**:

```
printf("Diametro = %f\n", circonferenza / 3.14);
```

Un compilatore **C** a **16** bit potrebbe anche decidere di inserire il valore **3.14** nel blocco **_CONST**; un compilatore **C** a **32** bit potrebbe invece decidere di inserire il valore **3.14** direttamente nel codice macchina (in formato **IEEE** per i numeri reali a **32** bit).
Siccome il blocco **_CONST** e' dotato di nome e attributi standard, per poterlo definire possiamo servirci della apposita direttiva semplificata:

```
.CONST
```

Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.CONST** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.
Il blocco **_CONST** viene sempre inserito nel gruppo **DGROUP** insieme ai blocchi **_DATA** e **_BSS**.

Lo stack di un programma puo' essere considerato come un blocco contenente dati non inizializzati; i compilatori che si servono dello stack creano un segmento di programma avente le seguenti caratteristiche:

```
_STACK SEGMENT PARA STACK USE16 'STACK'
```

All'interno di questo blocco e' presente un'area di memoria di dimensioni sufficienti per le esigenze di stack del programma; come gia' sappiamo questo blocco viene gestito dalla CPU attraverso la coppia **SS:SP**. In generale lo stack viene utilizzato per il salvataggio temporaneo di svariate informazioni; abbiamo anche visto che molti compilatori utilizzano lo stack per il passaggio degli argomenti alle procedure e per la creazione delle variabili locali.
Siccome il blocco **_STACK** e' dotato di nome e attributi standard, per poterlo definire possiamo servirci di una apposita direttiva semplificata che assume il seguente aspetto:

```
.STACK [stack_size]
```

Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.STACK** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.
Il parametro facoltativo **stack_size** se e' presente indica la dimensione in byte che vogliamo assegnare allo stack; e' fondamentale che venga specificato sempre un numero pari di byte in modo da garantire il corretto allineamento di **SP**. In assenza di questo parametro i compilatori utilizzano una dimensione predefinita che in genere e' pari a **1024** byte.
Il blocco **_STACK** viene normalmente inserito nel gruppo **DGROUP** insieme ai blocchi **_DATA**, **_BSS** e **_CONST**; se vogliamo tenere lo stack in un segmento separato, dobbiamo passare il parametro **FARSTACK** alla direttiva **.MODEL** Possiamo scrivere ad esempio:

```
.MODEL SMALL, PASCAL, FARSTACK
```

In assenza di questo parametro, l'assembler utilizza **NEARSTACK** come parametro predefinito; in questo caso il blocco stack viene inserito in **DGROUP**.

Se il blocco **_DATA** non e' sufficiente per contenere tutti i dati inizializzati del nostro programma, possiamo servirci di ulteriori blocchi di dati inizializzati; a tale proposito i compilatori utilizzano appositi segmenti di programma aventi le seguenti caratteristiche:

```
FAR_DATA SEGMENT PARA PRIVATE USE16 'FAR_DATA'
```

Come si puo' notare, questo blocco ha l'attributo di combinazione **PRIVATE**; questo significa che se definiamo numerosi blocchi **FAR_DATA** sia all'interno dello stesso modulo che in moduli differenti, tutti questi blocchi non verranno fusi tra loro. In pratica, se in un programma creiamo **10** blocchi **FAR_DATA**, otteniamo **10** blocchi **FAR_DATA** distinti; al posto di **FAR_DATA** possiamo utilizzare anche altri nomi purche' non si tratti di nomi riservati (come **_DATA** o **_BSS**).
Naturalmente, tutti i dati inizializzati presenti all'interno dei blocchi **FAR_DATA** vengono considerati di tipo **FAR**; per poter gestire questi dati e' necessario quindi specificare i loro indirizzi completi **seg:offset**. Tutto cio' implica che potrebbe presentarsi la necessita' di utilizzare **DS** per referenziare un blocco **FAR_DATA**; in casi del genere bisogna sempre ricordarsi di ripristinare il contenuto originario di **DS**. I compilatori utilizzano normalmente **DS** per gestire **DGROUP**; questo

significa che dopo aver utilizzato **DS** per referenziare un blocco **FAR_DATA**, dobbiamo ripristinarlo in modo da ottenere di nuovo **DS=DGROUP**.
Siccome i blocchi **FAR_DATA** sono dotati di attributi standard, per poterli definire possiamo servirci di una apposita direttiva semplificata che assume il seguente aspetto:
```
.FARDATA [nome]
```
Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.FARDATA** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.
Il parametro facoltativo **nome** se e' presente indica il nome che vogliamo utilizzare al posto di **FAR_DATA**; in assenza di questo parametro viene utilizzato il nome predefinito **FAR_DATA**.

Se il blocco **_BSS** non e' sufficiente per contenere tutti i dati non inizializzati del nostro programma, possiamo servirci di ulteriori blocchi di dati non inizializzati; a tale proposito i compilatori utilizzano appositi segmenti di programma aventi le seguenti caratteristiche:
```
FAR_BSS SEGMENT PARA PRIVATE USE16 'FAR_BSS'
```
Anche in questo caso notiamo che questo blocco ha l'attributo di combinazione **PRIVATE**; questo significa che se definiamo numerosi blocchi **FAR_BSS** sia all'interno dello stesso modulo che in moduli differenti, tutti questi blocchi non verranno fusi tra loro. Valgono in pratica tutte le considerazioni gia' svolte per i blocchi **FAR_DATA**; al posto di **FAR_BSS** possiamo utilizzare anche altri nomi purche' non si tratti di nomi riservati.
Tutti i dati non inizializzati presenti all'interno dei blocchi **FAR_BSS** vengono considerati di tipo **FAR**; per poter gestire questi dati e' necessario quindi specificare i loro indirizzi completi **seg:offset**.
Siccome i blocchi **FAR_BSS** sono dotati di attributi standard, per poterli definire possiamo servirci di una apposita direttiva semplificata che assume il seguente aspetto:
```
.FARDATA? [nome]
```
Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.FARDATA?** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.
Il parametro facoltativo **nome** se e' presente indica il nome che vogliamo utilizzare al posto di **FAR_BSS**; in assenza di questo parametro viene utilizzato il nome predefinito **FAR_BSS**.

Per il blocco codice principale i compilatori creano un segmento di programma avente le seguenti caratteristiche:
```
_TEXT SEGMENT WORD PUBLIC USE16 'CODE'
```
All'interno di questo blocco viene inserito il codice principale di un programma; in genere il blocco codice principale contiene anche l'**entry point**, per cui possiamo dire che al momento di caricare il programma in memoria il **SO** effettuera' l'inizializzazione **CS=_TEXT**.
Siccome il blocco **_CODE** e' dotato di nome e attributi standard, per poterlo definire possiamo servirci della apposita direttiva semplificata:
```
.CODE
```
Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.CODE** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.

Se il blocco **_TEXT** non e' sufficiente per contenere tutto il codice del programma, possiamo creare ulteriori blocchi di codice aventi le seguenti caratteristiche:
```
nome_TEXT SEGMENT WORD PUBLIC USE16 'CODE'
```
Per quanto riguarda il nome del segmento, la stringa **_TEXT** e' fissa, mentre la parte **nome** puo' essere scelta dal programmatore.
Siccome i blocchi **nome_TEXT** sono dotati di attributi standard, per poterli definire possiamo servirci di una apposita direttiva semplificata che assume il seguente aspetto:
```
.CODE [nome]
```

Quando l'assembler incontra questa direttiva, la espande e ottiene un segmento di programma avente il nome e gli attributi descritti in precedenza; la fine del blocco **.CODE** e' rappresentata dall'inizio del blocco successivo o dalla fine del modulo.
Il parametro facoltativo **nome** se e' presente indica il nome che vogliamo aggiungere alla stringa **_TEXT**; in assenza di questo parametro viene utilizzato il nome del modulo in cui si trova il blocco codice. Se ad esempio viene incontrata una direttiva **.CODE** in un modulo chiamato **LIB1**, si ottiene il nome **LIB1_TEXT**. Di conseguenza, se nel modulo **LIB1** sono presenti **10** blocchi **.CODE** privi del parametro **nome**, verranno creati **10** blocchi chiamati **LIB1_TEXT**; questi **10** blocchi hanno tutti lo stesso nome e gli stessi attributi (compreso **PUBLIC**), per cui verranno fusi in un unico blocco **LIB1_TEXT**. Se vogliamo avere **10** blocchi **nome_TEXT** distinti, dobbiamo inserirli in **10** moduli diversi (uno per modulo), oppure dobbiamo assegnare loro **10** nomi diversi.

E' necessario tener presente che non tutti i compilatori seguono le convenzioni appena esposte sui segmenti di programma standard; considerando ad esempio le numerose varianti del **BASIC** (compilato), puo' anche capitare che i diversi compilatori di questo linguaggio seguano convenzioni tra loro incompatibili. Per evitare questo problema, la cosa migliore da fare consiste nel servirsi delle direttive semplificate per i segmenti; in questo modo si delega all'assembler il compito di espandere queste direttive secondo criteri legati ai parametri che abbiamo specificato nella direttiva **.MODEL**. Si tenga presente inoltre che in presenza delle direttive semplificate per i segmenti, l'assembler provvede automaticamente a creare il gruppo **DGROUP**; se utilizziamo invece la forma classica per i segmenti standard, tutto questo lavoro spetta a noi. In questo caso corriamo anche il rischio di creare segmenti aventi caratteristiche incompatibili con il linguaggio con il quale vogliamo interfacciarci; in sostanza, la forma classica per i segmenti di programma dovrebbe essere utilizzata solo nel caso in cui sappiamo esattamente come dobbiamo comportarci.
Le caratteristiche avanzate di **MASM** e **TASM** permettono di supportare tutte le convenzioni appena illustrate sui segmenti di programma; vengono supportate quindi le direttive come **.DATA**, **.CODE**, etc, e viene creato automaticamente il gruppo **DGROUP**. Nel caso generale l'assembler pone:

```
DGROUP GROUP _DATA, _CONST, _STACK, _BSS
```

Se vogliamo che lo stack venga tenuto distinto da **DGROUP**, dobbiamo passare alla direttiva **.MODEL** il parametro **FARSTACK**.
L'aspetto piu' importante da ricordare e' che in **Assembly** come al solito l'inizializzazione dei registri di segmento per i dati come **DS** o **ES** spetta al programmatore; in generale, la cosa migliore da fare consiste nel seguire le convenzioni dei compilatori che pongono **DS=DGROUP**.
Gli assembler che supportano tutte queste caratteristiche avanzate, creano automaticamente una serie di variabili simboliche che il programmatore puo' utilizzare all'interno dei programmi; in particolare, vengono create le seguenti variabili:

La variabile **@data** rappresenta il segmento di programma correntemente referenziato da **DS**; se **DS=DGROUP** abbiamo chiaramente **@data=DGROUP**.

La variabile **@stack** rappresenta il segmento di programma correntemente referenziato da **SS**; se **SS=_STACK** abbiamo chiaramente **@stack=_STACK**.

La variabile **@fardata** rappresenta il segmento **FAR_DATA** corrente; se ci troviamo nel blocco codice di un modulo dotato di un blocco di dati **FAR** chiamato **LIB1_DATA**, in quel preciso istante si ha: **@fardata=LIB1_DATA**.

La variabile **@fardata?** rappresenta il segmento **FAR_BSS** corrente; se ci troviamo nel blocco codice di un modulo dotato di un blocco di dati **FAR** chiamato **LIB1_BSS**, in quel preciso istante si ha:
@fardata?=LIB1_BSS.

La variabile **@code** rappresenta il segmento di programma correntemente referenziato da **CS**; se il **SO** inizializza **CS** ponendo **CS=_TEXT**, in quel preciso momento si ha: **@code=_TEXT**. Se in seguito saltiamo ad un blocco codice **LIB1_TEXT**, in quel preciso momento si ha:
@code=LIB1_TEXT.

Per illustrare in pratica tutti questi concetti e' necessario analizzare in dettaglio i vari casi che si possono presentare in relazione ai diversi modelli di memoria; vediamo quindi una serie di esempi pratici che si riferiscono all'uso delle caratteristiche avanzate di **MASM** e **TASM** nei programmi interamente scritti in **Assembly**. Nei capitoli successivi viene analizzato invece l'uso delle caratteristiche avanzate di **MASM** e **TASM** nel processo di interfacciamento tra l'**Assembly** e i linguaggi di alto livello.
Prima di proseguire e' necessario ricordare l'esistenza di un tristemente famoso bug del **MASM** che si manifesta in presenza della direttiva **GROUP**; il bug consiste nel fatto che l'operatore **OFFSET** applicato ad una variabile appartenente ad un gruppo, calcola il relativo offset rispetto al segmento di appartenenza della variabile e non rispetto al gruppo. Se abbiamo ad esempio una varibile **var1** definita nel segmento **_DATA** che appartiene al gruppo **DGROUP**, allora l'istruzione:
```
mov bx, offset var1
```
carica erroneamente in **BX** l'offset di **var1** calcolato rispetto a **_DATA** e non rispetto a **DGROUP**; per aggirare questo problema si puo' scrivere:
```
mov bx, offset DGROUP:var1
```
oppure:
```
lea bx, var1
```
Come e' stato detto in un precedente capitolo, questo bug e' presente solo nelle vecchissime versioni del **MASM**; le versioni recenti di questo assembler non soffrono piu' di questo problema. Il **TASM** funziona in modalita' predefinita di emulazione del **MASM** e per una assurda decisione dei programmatori della **Borland**, simula anche il bug del **MASM**; la situazione grottesca che si viene a creare e' data quindi dal fatto che chi usa **TASM** deve preoccuparsi del bug citato in precedenza, mentre chi usa le versioni recenti del **MASM** non ha piu' questa preoccupazione.

28.6 Modello TINY

La struttura classica di un programma con modello di memoria **TINY** prevede la presenza di un solo blocco contenente codice, dati e stack; possiamo suddividere se necessario tutte queste informazioni in **_DATA**, **_CONST**, **_BSS**, **_STACK** e **_TEXT**. L'assembler gestisce tutta questa situazione creando automaticamente il gruppo:
```
DGROUP GROUP _TEXT, _DATA, _CONST, _STACK, _BSS
```
In base a questo schema possiamo dire che tutti i dati e tutte le procedure del programma possono essere gestiti attraverso indirizzi di tipo **NEAR**; se viene generato un programma in formato **COM**, al momento del suo caricamento in memoria il **SO** pone:
CS = DS = ES = SS = DGROUP.
Se invece viene generato un programma in formato **EXE**, al momento del suo caricamento in memoria il **SO** pone:
CS = _TEXT = DGROUP e **SS = _STACK = DGROUP**.
Il programmatore da parte sua deve porre:
DS = DGROUP (eventualmente possiamo assegnare **DGROUP** anche a **ES**).

Per analizzare in pratica tutti questi concetti, vediamo un esempio del tutto simile a quello illustrato nelle Figure 18, 19 e 20 del precedente capitolo; in questo caso vediamo come si applica la sintassi avanzata di **MASM** e **TASM** per la generazione di un eseguibile in formato **COM**.
Prima di tutto definiamo il modulo **FILELIB1.ASM** contenente la procedura **setCursor**; questo modulo viene illustrato dalla Figura 7.

Figura 7 - Modulo FILELIB1.ASM

```
; File filelib1.asm

.MODEL      TINY, C
.386                               ; set di istruzioni a 32 bit

.DATA

   PUBLIC   stringa1

stringa1    db    'stringa1 definita in FILELIB1', 10, 13, '$'

.CODE

   PUBLIC   setCursor

; void setCursor(int riga, int colonna)

setCursor proc riga: BYTE, colonna: BYTE

   mov      ah, 02h                ; ah = Set Cursor Position
   mov      bh, 00h                ; bh = pagina video 0
   mov      dh, riga               ; dh = riga
   mov      dl, colonna            ; dl = colonna
   int      10h                    ; chiama i servizi Video BIOS

   ret                             ; return

setCursor endp

   END
```

Come si puo' notare, stiamo utilizzando il modello **TINY** e le convenzioni **C** per le procedure; vediamo anche che all'interno di **setCursor** e' scomparso tutto il codice relativo al **prolog code** e all'**epilog code**. Un'altro aspetto importante da notare e' dato dal fatto che la sintassi avanzata di **MASM** e **TASM** ci permette di dichiarare parametri di tipo **BYTE** per le procedure; naturalmente l'assembler inserira' ugualmente nello stack valori a **16** bit contenenti negli **8** bit meno significativi l'argomento da passare alla procedura. Un'altra novita' e' rappresentata dalla scomparsa delle fastidiose direttive **ASSUME**; anche questo aspetto viene gestito infatti direttamente dall'assembler.

Passiamo ora al modulo **FILELIB2.ASM** contenente la procedura **printStr** che si serve anche di **setCursor**; questo modulo viene illustrato dalla Figura 8.

Figura 8 - Modulo FILELIB2.ASM

```
; File filelib2.asm

.MODEL      TINY, C
.386                           ; set di istruzioni a 32 bit

setCursor   PROTO :BYTE, :BYTE   ; prototipo di setCursor

.DATA

   PUBLIC   stringa2

stringa2    db    'stringa2 definita in FILELIB2', 10, 13, '$'

.CODE

   PUBLIC   printStr

; void printStr(int riga, int colonna, char *strOffs)

printStr proc riga: BYTE, colonna: BYTE, strOffs: WORD

   call     setCursor, word ptr riga, word ptr colonna

   mov      dx, strOffs          ; ds:dx punta alla stringa
   mov      ah, 09h              ; servizio Display String
   int      21h                  ; chiama i servizi DOS

   ret                           ; return

printStr endp

   END
```

Nel modulo **FILELIB2.ASM** notiamo la presenza del prototipo di **setCursor** che funge anche da direttiva **EXTRN** per la stessa procedura; se si vuole utilizzare la direttiva **EXTRN** si consiglia di inserirla al di fuori di qualsiasi segmento di programma (preferibilmente nella stessa posizione in cui in Figura 8 si trova il prototipo di **setCursor**).
Naturalmente, se si sta utilizzando **MASM** bisogna sostituire la direttiva **CALL** con la direttiva **INVOKE**; in questo caso e' anche necessario eliminare gli operatori **WORD PTR** (richiesti da **TASM**) nella chiamata di **setCursor**.
La procedura **printStr** visualizza la stringa che ha ricevuto per indirizzo attraverso il parametro **strOffs**; siccome ci troviamo nel modello **TINY**, possiamo gestire tutti i dati del programma attraverso la sola componente **offset**. Osserviamo infatti che **printStr** pone **DX=strOffs** e lascia inalterato **DS**; non e' necessario modificare **DS** in quanto si ha **DS=DGROUP** per tutta la fase di esecuzione del programma.

Passiamo infine al modulo principale **FILEMAIN.ASM** del programma; questo modulo viene illustrato dalla Figura 9.

Figura 9 - Modulo FILEMAIN.ASM

```
; File filemain.asm

.MODEL      TINY, C
.386                            ; set di istruzioni a 32 bit

setCursor   PROTO :BYTE, :BYTE
printStr    PROTO :BYTE, :BYTE, :WORD

   EXTRN    stringa1: BYTE, stringa2: BYTE

.DATA

stringa3    db    'stringa3 definita in FILEMAIN', 10, 13, '$'

.CODE

   org      0100h               ; 256 byte per il PSP

start:                          ; entry point

   mov      byte ptr stringa1[0], 'S'
   call     printStr, 10, 0, offset stringa1
   mov      byte ptr stringa2[0], 'S'
   call     printStr, 12, 0, offset stringa2
   mov      byte ptr stringa3[0], 'S'
   call     printStr, 14, 0, offset stringa3

   xor      al, al                    ; exit code
   mov      ah, 4Ch                   ; terminate process
   int      21h                       ; servizi DOS

   END      start
```

Nel modulo principale notiamo come al solito la presenza della direttiva **ORG** che libera **256** byte per il **PSP**; il modulo principale deve essere l'unico a possedere l'entry point del programma. Il compito di predisporre lo stack spetta come sappiamo al **SO**; se invece vogliamo generare un eseguibile in formato **EXE**, questo compito spetta a noi (come viene illustrato negli esempi relativi agli altri modelli di memoria).
Nel blocco delle istruzioni principali notiamo che ogni stringa da visualizzare, viene modificata convertendo in maiuscolo la sua iniziale (da **'s'** a **'S'**); anche queste istruzioni mostrano che nel modello **TINY** possiamo operare sulla sola componente **offset** dei dati del programma. Se si sta utilizzando il **MASM** bisogna ricordarsi di sostituire le direttive **CALL** con le direttive **INVOKE**.

Se in fase di assemblaggio richiediamo la generazione dei **listing files** di questi moduli, possiamo analizzare tutto il lavoro svolto dall'assembler; nella parte riservata al codice macchina notiamo subito che l'assembler ha generato per le procedure il **prolog code** e l'**epilog code** conforme alle convenzioni **C**.
Passando alla **symbol table** vediamo che l'assembler ha assegnato l'attributo **extern** alle procedure esterne per le quali abbiamo specificato il prototipo; sempre nella **symbol table** si nota la presenza delle variabili simboliche **@code**, **@data**, etc, create dall'assembler. Nel caso particolare del modello di memoria **TINY** si puo' constatare che si ha ovviamente:
@code = @data = @stack = DGROUP.

Nella parte finale della **symbol table** si nota poi che i blocchi **_DATA** e **_TEXT** del programma sono stati inseriti in un gruppo chiamato proprio **DGROUP**.
Attraverso il **map file** del programma possiamo anche constatare che il linker ha fuso i tre blocchi **_DATA** presenti nei tre moduli, in un unico blocco **_DATA**; analogamente, i tre blocchi **_TEXT** presenti nei tre moduli vengono fusi in un unico blocco **_TEXT**. Si tenga presente che il **map file** mostra separatamente tutti i segmenti di programma anche se fanno parte di **DGROUP**.
Osserviamo infine che se vogliamo convertire tutto il programma precedente in versione **Pascal**, dobbiamo semplicemente inserire il parametro **PASCAL** al posto di **C** nelle direttive **.MODEL** dei tre moduli.

28.7 Modello SMALL

La struttura classica di un programma con modello di memoria **SMALL** prevede la presenza di un solo blocco dati (compreso lo stack) e di un solo blocco codice; possiamo suddividere se necessario tutte queste informazioni in **_DATA**, **_CONST**, **_BSS**, **_STACK** e **_TEXT**. L'assembler gestisce tutta questa situazione creando automaticamente il gruppo:

```
DGROUP GROUP _DATA, _CONST, _STACK, _BSS
```

Il blocco **_TEXT** questa volta viene escluso dal gruppo **DGROUP**; in base a questo schema possiamo dire che tutti i dati e tutte le procedure del programma possono essere gestiti attraverso indirizzi di tipo **NEAR**.
In avvio del programma il **SO** pone:
CS = _TEXT e **SS = _STACK = DGROUP**.
Il programmatore da parte sua deve porre:
DS = DGROUP.
Se vogliamo tenere separato lo stack da **DGROUP**, dobbiamo passare il parametro **FARSTACK** alla direttiva **.MODEL**; in assenza di questo parametro, l'assembler segue le convenzioni dei compilatori inserendo lo stack in **DGROUP**. Questo stesso comportamento puo' essere ottenuto passando il parametro **NEARSTACK** alla direttiva **.MODEL**; le cosiderazioni appena esposte sono valide per tutti gli altri modelli di memoria illustrati nel seguito del capitolo.

Se vogliamo adattare al modello **SMALL** l'esempio precedente, la prima cosa da fare consiste nell'inserire in tutti i tre moduli la direttiva:
```
.MODEL SMALL, C
```
A questo punto possiamo procedere con le altre modifiche da apportare ai tre moduli; i due moduli **FILELIB1.ASM** e **FILELIB2.ASM** coincidono esattamente con quelli di Figura 7 e di Figura 8.
Il modulo **FILEMAIN.ASM** subisce le modifiche visibili in Figura 10.

Figura 10 - Modulo FILEMAIN.ASM

```
; File filemain.asm

.MODEL      SMALL, C
.386                            ; set di istruzioni a 32 bit

setCursor   PROTO :BYTE, :BYTE
printStr    PROTO :BYTE, :BYTE, :WORD

   EXTRN    stringa1: BYTE, stringa2: BYTE

.STACK      1024                ; 1024 byte per lo stack

.DATA

stringa3    db    'stringa3 definita in FILEMAIN', 10, 13, '$'

.CODE

start:                          ; entry point

   mov      ax, DGROUP          ; carica DGROUP
   mov      ds, ax              ; in ds

   mov      byte ptr stringa1[0], 'S'
   call     printStr, 10, 0, offset stringa1
   mov      byte ptr stringa2[0], 'S'
   call     printStr, 12, 0, offset stringa2
   mov      byte ptr stringa3[0], 'S'
   call     printStr, 14, 0, offset stringa3

   xor      al, al                    ; exit code
   mov      ah, 4Ch                   ; terminate process
   int      21h                       ; servizi DOS

   END      start
```

La prima novita' da notare riguarda la presenza della direttiva **.STACK**; questa direttiva ci permette di riservare **1024** byte di memoria per le variabili temporanee. Sempre in Figura 10 notiamo anche la scomparsa della direttiva **ORG**; subito dopo l'etichetta **start** inoltre e' presente l'inizializzazione di **DS**.
All'interno del blocco codice principale possiamo constatare che le variabili del programma vengono gestite attraverso il loro offset; questo perche' esiste un solo blocco dati **DGROUP**, e quindi **DS** rimane invariato per tutta la fase di esecuzione del programma.
In una situazione di questo genere l'assembler crea le tre variabili simboliche **@code**, **@data** e **@stack**; le prime due variabili valgono:
@code = _TEXT, **@data = DGROUP**.
Per inizializzare **DS** in Figura 10 possiamo usare quindi indifferentemente **DGROUP** o **@data**.
In presenza del parametro **NEARSTACK** l'assembler pone: **@stack = DGROUP**.
In presenza invece del parametro **FARSTACK** l'assembler pone: **@stack = _STACK**.

Per giustificare questa situazione possiamo consultare il **map file** constatando che il linker ha fuso i tre blocchi **_DATA** presenti nei tre moduli, in un unico blocco **_DATA**; analogamente, i tre blocchi **_TEXT** presenti nei tre moduli vengono fusi in un unico blocco **_TEXT**. Alla fine si ottiene un solo blocco **_TEXT**, un solo blocco **_DATA** e un solo blocco **_STACK**.

28.8 Modello MEDIUM

La struttura classica di un programma con modello di memoria **MEDIUM** prevede la presenza di un solo blocco dati (compreso lo stack) e di due o piu' blocchi di codice; possiamo suddividere se necessario tutte queste informazioni in **_DATA**, **_CONST**, **_BSS**, **_STACK**, **_TEXT** e uno o piu' **nome_TEXT**. L'assembler gestisce tutta questa situazione creando automaticamente il gruppo:

```
DGROUP GROUP _DATA, _CONST, _STACK, _BSS
```

Il blocco **_TEXT** non viene inserito in **DGROUP**; gli eventuali blocchi **nome_TEXT** che differiscono per il **nome**, non verranno fusi tra loro. In base a questo schema possiamo dire che tutti i dati possono essere gestiti attraverso indirizzi di tipo **NEAR**, mentre le procedure devono essere necessariamente di tipo **FAR**.
In avvio del programma il **SO** pone:
CS = _TEXT e **SS = _STACK = DGROUP**.
Il programmatore da parte sua deve porre:
DS = DGROUP.

Se vogliamo adattare al modello **MEDIUM** l'esempio precedente, la prima cosa da fare consiste nell'inserire in tutti i tre moduli la direttiva:

```
.MODEL MEDIUM, C
```

A questo punto possiamo procedere con le altre modifiche da apportare ai tre moduli; i due moduli **FILELIB1.ASM** e **FILELIB2.ASM** coincidono esattamente con quelli di Figura 7 e di Figura 8.
Al momento dell'assemblaggio di questi due moduli, l'assembler nota il modello di memoria **MEDIUM**, e in presenza delle direttive **.CODE** produce:

un blocco codice chiamato **FILELIB1_TEXT** per il modulo **FILELIB1**;
un blocco codice chiamato **FILELIB2_TEXT** per il modulo **FILELIB2**.
Il modulo **FILEMAIN.ASM** risulta identico a quello di Figura 10; quando l'assembler incontra la direttiva **.CODE** di **FILEMAIN.ASM**, produce un blocco codice chiamato **_TEXT** che rappresenta il blocco codice principale. Si tenga presente che il **TASM** produce invece un blocco codice chiamato **FILEMAIN_TEXT**; in ogni caso questo blocco contiene l'**entry point** per cui il **SO** lo usera' per inizializzare **CS**.
In presenza di questa situazione l'assembler crea le variabili simboliche: **@data**, **@stack** e **@code**; quando ci troviamo all'interno del blocco **_TEXT** (o **FILEMAIN_TEXT**), l'assembler pone:
@code = _TEXT, **@data = DGROUP**.
Quando ci troviamo all'interno del blocco **FILELIB1_TEXT**, l'assembler pone:
@code = FILELIB1_TEXT, **@data = DGROUP**.
Quando ci troviamo all'interno del blocco **FILELIB2_TEXT**, l'assembler pone:
@code = FILELIB2_TEXT, **@data = DGROUP**.
Inoltre, in presenza del parametro **NEARSTACK** si ha: **@stack = DGROUP**.
In presenza invece del parametro **FARSTACK** si ha: **@stack = _STACK**.
Per giustificare questa situazione possiamo consultare il **map file** constatando che il linker ha fuso i tre blocchi **_DATA** presenti nei tre moduli, in un unico blocco **_DATA**; i tre blocchi **_TEXT**, **FILELIB1_TEXT** e **FILELIB2_TEXT** restano invece distinti. Alla fine si ottiene un blocco **_DATA**, un blocco **_STACK**, un blocco **_TEXT**, un blocco **FILELIB1_TEXT** e un blocco **FILELIB2_TEXT**.

28.9 Modello COMPACT

La struttura classica di un programma con modello di memoria **COMPACT** prevede la presenza di un solo blocco codice e di due o piu' blocchi di dati; possiamo suddividere se necessario tutte queste informazioni in **_DATA**, **_CONST**, **_BSS**, **_STACK**, uno o piu' **FAR_DATA**, uno o piu' **FAR_BSS** e **_TEXT**. L'assembler gestisce tutta questa situazione creando automaticamente il gruppo:

```
DGROUP GROUP _DATA, _CONST, _STACK, _BSS
```

Il blocco **_TEXT** non viene inserito in **DGROUP**; gli eventuali blocchi **FAR_DATA** e **FAR_BSS** restano tutti distinti e non fanno parte di **DGROUP**. In base a questo schema possiamo dire che tutte le procedure possono essere di tipo **NEAR**, mentre i dati devono essere gestiti necessariamente con indirizzi di tipo **FAR**.
In avvio del programma il **SO** pone:
CS = _TEXT e **SS = _STACK = DGROUP**.
Il programmatore da parte sua deve porre:
DS = DGROUP.
Se vogliamo adattare al modello **COMPACT** l'esempio precedente, la prima cosa da fare consiste nell'inserire in tutti i tre moduli la direttiva:

```
.MODEL COMPACT, C
```

A questo punto possiamo procedere con le altre modifiche da apportare ai tre moduli; all'interno del modulo **FILELIB1.ASM** il blocco **.DATA** diventa:

```
.FARDATA

   PUBLIC   stringa1

stringa1    db    'stringa1 definita in FILELIB1', 10, 13, '$'
```

All'interno del modulo **FILELIB2.ASM** il blocco **.DATA** diventa:

```
.FARDATA

   PUBLIC   stringa2

stringa2    db    'stringa2 definita in FILELIB2', 10, 13, '$'
```

Inoltre la procedura **printStr** ha bisogno questa volta dell'indirizzo completo **seg:offset** della stringa da visualizzare; la definizione di questa procedura diventa quindi:

```
; void printStr(int riga, int colonna, char far *strAddr)
; strAddr = strSeg + strOffs

printStr proc riga: BYTE, colonna: BYTE, strOffs: WORD, strSeg: WORD

   call     setCursor, word ptr riga, word ptr colonna

   push     ds                 ; preserva ds
   mov      ds, strSeg         ; ds = strSeg
   mov      dx, strOffs        ; ds:dx punta alla stringa
   mov      ah, 09h            ; servizio Display String
   int      21h                ; chiama i servizi DOS
   pop      ds                 ; ripristina ds

   ret                         ; return

printStr endp
```

Come possiamo notare, questa volta **printStr** ha bisogno di conoscere anche la componente **seg** della stringa da visualizzare; a tale proposito si puo' notare la presenza del quarto parametro **strSeg**. Osserviamo anche che in questa situazione, **printStr** per poter chiamare il servizio **Display String** deve modificare anche **DS**; e' fondamentale quindi che il contenuto di questo registro di segmento venga preservato.

Il modulo **FILEMAIN.ASM** subisce le modifiche visibili in Figura 11.

Figura 11 - Modulo FILEMAIN.ASM

```
; File filemain.asm

.MODEL      COMPACT, C
.386                             ; set di istruzioni a 32 bit

setCursor   PROTO :BYTE, :BYTE
printStr    PROTO :BYTE, :BYTE, :WORD, :WORD

   EXTRN    stringa1: BYTE, stringa2: BYTE

.STACK      1024                 ; 1024 byte per lo stack

.DATA

stringa3    db    'stringa3 definita in FILEMAIN', 10, 13, '$'

.CODE

start:                           ; entry point

   mov      ax, DGROUP           ; carica DGROUP
   mov      ds, ax               ; in ds

   mov      ax, seg stringa1
   mov      es, ax
   mov      byte ptr es:[stringa1], 'S'
   call     printStr, 10, 0, offset stringa1, es

   mov      ax, seg stringa2
   mov      es, ax
   mov      byte ptr es:[stringa2], 'S'
   call     printStr, 12, 0, offset stringa2, es

   mov      byte ptr [stringa3], 'S'
   call     printStr, 14, 0, offset stringa3, ds

   xor      al, al                     ; exit code
   mov      ah, 4Ch                    ; terminate process
   int      21h                        ; servizi DOS

   END      start
```

Questa volta abbiamo a che fare con stringhe definite in tre segmenti di dati differenti, per cui siamo costretti a lavorare con gli indirizzi **FAR** per i dati; siccome stiamo utilizzando **DS** per referenziare **DGROUP**, possiamo utilizzare **ES** per accedere ai due blocchi **FAR_DATA**. Il primo gruppo di istruzioni principali carica appunto in **ES** il segmento di appartenenza di **stringa1**; a questo punto viene modificata la lettera iniziale della stringa con l'istruzione:

```
mov byte ptr es:[stringa1], 'S'
```

Si puo' anche scrivere:
```
mov byte ptr es:stringa1[0], 'S'
```
Questo lavoro viene ripetuto anche per **stringa2**.
Per quanto riguarda **stringa3**, possiamo notare che questa stringa si trova in **DGROUP**; se siamo sicuri che **DS=DGROUP**, allora possiamo accedere ad essa con un indirizzo di tipo **NEAR** (vedere la Figura 11). Quando la CPU incontra l'offset di **stringa3**, utilizza automaticamente **DS** come registro dati predefinito; nella successiva chiamata di **printStr**, come quarto parametro, al posto di **DS** possiamo anche passare **@data** o **DGROUP**.
La procedura **printStr** puo' essere modificata in modo da sostituire il terzo e il quarto parametro di tipo **WORD** con un solo parametro di tipo **DWORD**; in questo parametro possiamo inserire l'intera coppia **seg:offset** della stringa da passare alla procedura. A tale proposito, possiamo anche definire una apposita variabile a **32** bit in un blocco dati non inizializzati nel modulo **FILEMAIN.ASM**; possiamo scrivere ad esempio:

```
.DATA?

farAddr     dd    ?
```

Questa variabile puo' essere utilizzata per contenere il parametro **seg:offset** da passare a **printStr**; possiamo scrivere ad esempio:

```
mov      word ptr farAddr[0], offset stringa1
mov      word ptr farAddr[2], seg stringa1
call     printStr, 10, 0, farAddr
```

Come si puo' notare, e' importante ricordarsi di seguire le convenzioni **Intel** per gli indirizzi **FAR**; bisogna inserire cioe' la componente **offset** nella word meno significativa di **farAddr**. Utilizzando questo metodo, la procedura **printStr** deve essere modificata in questo modo:

```
; void printStr(int riga, int colonna, char far *strAddr)

printStr proc riga: BYTE, colonna: BYTE, strAddr :DWORD

   call     setCursor, word ptr riga, word ptr colonna

   push     ds                      ; preserva ds
   mov      ds, word ptr strAddr[2] ; ds = strSeg
   mov      dx, word ptr strAddr[0] ; ds:dx punta alla stringa
   mov      ah, 09h                 ; servizio Display String
   int      21h                     ; chiama i servizi DOS
   pop      ds                      ; ripristina ds

   ret                              ; return

printStr endp
```

Il parametro **strAddr** a **32** bit puo' essere caricato nella coppia **DS:DX** anche con l'unica istruzione **LDS**; in questo caso dobbiamo scrivere:
```
lds dx, strAddr
```
Naturalmente le istruzioni come **LDS**, **LES**, **LFS**, etc, funzionano in modo corretto solo se l'operando sorgente (come **strAddr**) contiene una coppia **seg:offset** che rispetta le convenzioni **Intel**.

E' importante ricordare che i blocchi **FAR_DATA** e **FAR_BSS** hanno attributo di combinazione **PRIVATE** per cui restano tutti distinti; in presenza di questa situazione l'assembler crea le variabili simboliche: **@data**, **@fardata**, **@stack** e **@code**. Quando ci troviamo all'interno del blocco **_TEXT** di **FILEMAIN**, l'assembler pone:
@code = _TEXT, **@data = DGROUP**.
Quando ci troviamo all'interno del blocco **_TEXT** di **FILELIB1**, l'assembler pone:
@code = _TEXT, **@data = DGROUP**, **@fardata = FAR_DATA** (di **FILELIB1**).
Quando ci troviamo all'interno del blocco **_TEXT** di **FILELIB2**, l'assembler pone:
@code = _TEXT, **@data = DGROUP**, **@fardata = FAR_DATA** (di **FILELIB2**).

Inoltre, in presenza del parametro **NEARSTACK** si ha: **@stack = DGROUP**.
In presenza invece del parametro **FARSTACK** si ha: **@stack = _STACK**.

Per giustificare questa situazione possiamo consultare il **map file** constatando che il linker ha fuso i tre blocchi **_TEXT** presenti nei tre moduli, in un unico blocco **_TEXT**; i tre blocchi **_DATA**, **FAR_DATA** di **FILELIB1** e **FAR_DATA** di **FILELIB2** restano invece distinti. Alla fine si ottiene un blocco **_DATA**, due blocchi **FAR_DATA**, un blocco **_STACK** e un blocco **_TEXT**.

28.10 Modelli LARGE e HUGE

La struttura classica di un programma con modello di memoria **LARGE** prevede la presenza di due o piu' blocchi di codice e di due o piu' blocchi di dati; possiamo suddividere se necessario tutte queste informazioni in **_DATA**, **_CONST**, **_BSS**, **_STACK**, uno o piu' **FAR_DATA**, uno o piu' **FAR_BSS**, **_TEXT** e uno o piu' **nome_TEXT**. L'assembler gestisce tutta questa situazione creando automaticamente il gruppo:
```
DGROUP GROUP _DATA, _CONST, _STACK, _BSS
```
Il blocco **_TEXT** non viene inserito in **DGROUP**; gli eventuali blocchi **FAR_DATA**, **FAR_BSS** e **nome_TEXT** restano tutti distinti e non fanno parte di **DGROUP**. In base a questo schema possiamo dire che tutte le procedure devono essere di tipo **FAR**, e anche i dati devono essere gestiti necessariamente con indirizzi di tipo **FAR**.
In avvio del programma il **SO** pone:
CS = _TEXT e **SS = _STACK**.
Il programmatore da parte sua deve porre:
DS = DGROUP.

Se vogliamo adattare al modello **LARGE** l'esempio precedente, la prima cosa da fare consiste nell'inserire in tutti i tre moduli la direttiva:
```
.MODEL LARGE, C
```
A questo punto possiamo procedere con le altre modifiche da apportare ai tre moduli. I blocchi di dati presenti nei tre tre moduli coincidono esattamente con quelli utilizzati per il modello **COMPACT**; abbiamo quindi un blocco **_DATA** in **FILEMAIN.ASM**, un blocco **FAR_DATA** in **FILELIB1.ASM** e un blocco **FAR_DATA** in **FILELIB2.ASM**. I blocchi di codice dei tre tre moduli coincidono esattamente con quelli utilizzati per il modello **MEDIUM**; abbiamo quindi un blocco **_TEXT** in **FILEMAIN.ASM**, un blocco **FILELIB1_TEXT** in **FILELIB1.ASM** e un blocco **FILELIB2_TEXT** in **FILELIB2.ASM**.
Il linker produce quindi un programma contenente un blocco **_DATA**, un blocco **_STACK**, due blocchi distinti **FAR_DATA**, un blocco **_TEXT**, un blocco **FILELIB1_TEXT** e un blocco **FILELIB2_TEXT**; a tale proposito si puo' consultare il **map file** del programma. Come al solito si deve tener presente che il **TASM** produce un blocco **FILEMAIN_TEXT** al posto di **_TEXT**.

In presenza di questa situazione l'assembler crea le variabili simboliche: **@data**, **@fardata**, **@stack** e **@code**. Quando ci troviamo all'interno del blocco **_TEXT** di **FILEMAIN** l'assembler pone:
@code = _TEXT, **@data = DGROUP**.
Quando ci troviamo all'interno del blocco **FILELIB1_TEXT**, l'assembler pone:
@code = FILELIB1_TEXT, **@data = DGROUP**, **@fardata = FAR_DATA** (del modulo **FILEMAIN1**).
Quando ci troviamo all'interno del blocco **FILELIB2_TEXT**, l'assembler pone:
@code = FILELIB2_TEXT, **@data = DGROUP**, **@fardata = FAR_DATA** (del modulo **FILEMAIN2**).

Inoltre, in presenza del parametro **NEARSTACK** si ha: **@stack = DGROUP**.
In presenza invece del parametro **FARSTACK** si ha: **@stack = _STACK**.

Il modello **HUGE** come sappiamo e' del tutto simile al modello **LARGE**; i compilatori che supportano il modello **HUGE**, lo utilizzano per dare la possibilita' al programmatore di definire singoli dati che superano la dimensione di **64** Kb.

28.11 Modello FLAT

I segmenti convenzionali utilizzati nel modello **FLAT** sono del tutto simili a quelli che vengono impiegati negli altri modelli di memoria; le differenze sostanziali riguardano l'attributo **SIZE** che e' naturalmente **USE32**, e l'attributo **ALIGN** che diventa **DWORD** (allineamento ottimale per le CPU a **32** bit).
Gli unici blocchi esistenti nel modello **FLAT** sono: **_DATA**, **_BSS**, **_CONST**, **_STACK** e **_TEXT**; i blocchi **_DATA**, **_BSS**, **_CONST** e **_STACK**, come al solito vengono inseriti in un gruppo chiamato **DGROUP**. L'eseguibile che si ottiene e' formato da un blocco **DGROUP** per i dati e da un blocco **_TEXT** per il codice; si tratta quindi di una situazione formalmente identica al caso del modello **SMALL**.
I segmenti di dati **FAR_DATA** e **FAR_BSS**, e i segmenti di codice **nome_TEXT** perdono significato; non bisogna dimenticare infatti che nel modello **FLAT** ogni programma viene inserito in un segmento virtuale da **4** Gb (**flat segment**) all'interno del quale vengono sistemati i dati, il codice e lo stack. Per muoverci all'interno di un **flat segment** utilizziamo indirizzi **NEAR** rappresentati da offset a **32** bit che possono assumere tutti i valori compresi tra **00000000h** e **FFFFFFFFh**; in questo modo e' possibile esplorare tutto il segmento virtuale da **4** Gb.
Generalmente, i linker che supportano il modello **FLAT** dei **SO** come **Windows** a **32** bit, **OS/2**, etc, provvedono anche a predisporre lo stack del programma; in definitiva possiamo dire quindi che nel modello **FLAT** avremo i seguenti segmenti di programma:

Per i dati inizializzati:
```
_DATA SEGMENT DWORD PUBLIC USE32 'DATA'
```
La direttiva semplificata per questo segmento e' **.DATA**

Per i dati non inizializzati:
```
_BSS SEGMENT DWORD PUBLIC USE32 'BSS'
```
La direttiva semplificata per questo segmento e' **.DATA?**

Per i dati costanti:
```
_CONST SEGMENT DWORD PUBLIC USE32 'CONST'
```
La direttiva semplificata per questo segmento e' **.CONST**

Per il codice:
```
_TEXT SEGMENT DWORD PUBLIC USE32 'CODE'
```

La direttiva semplificata per questo segmento e' **.CODE**

Per maggiori dettagli sul modello **FLAT** si puo' fare riferimento alla sezione **Win32 Assembly**.

28.12 Le istruzioni ENTER e LEAVE

In precedenza e' stato detto che nel caso particolare del **TASM**, e' possibile richiedere la generazione di un dettagliato **listing file** che illustra anche il lavoro compiuto dall'assembler in relazione all'uso delle caratteristiche avanzate; in questo modo, assemblando i precedenti esempi con la direttiva **.8086**, possiamo constatare che il **listing file** che si ottiene mostra chiaramente che il **TASM** ha convertito le varie direttive avanzate nel classico codice necessario per la gestione dei parametri e delle variabili locali di una procedura.
Nel caso generale di una procedura dotata di parametri e di **n** byte di variabili locali, il codice di "ingresso" e' costituito dalle seguenti istruzioni:

```
push     bp       ; preserva bp
mov      bp, sp   ; ss:bp = ss:sp
sub      sp, n    ; spazio per le var. locali
```

Come gia' sappiamo, grazie a queste istruzioni possiamo trovare i parametri della procedura a spiazzamenti positivi rispetto a **BP**, e le variabili locali a spiazzamenti negativi rispetto a **BP**; la fase appena descritta rappresenta la creazione del cosiddetto **stack frame** della procedura.
Al termine della stessa procedura, il codice di "uscita" e' costituito dalle seguenti istruzioni:

```
mov      sp, bp   ; pulizia var. locali
pop      bp       ; ripristina bp
```

Lo scopo di queste istruzioni e' quello di ripristinare **SP** e **BP** prima di restituire il controllo al caller; la fase appena descritta rappresenta la distruzione dello **stack frame** della procedura che comprende anche la rimozione degli argomenti dallo stack (che puo' competere al caller o alla procedura stessa).
Se ora proviamo ad abilitare il set di istruzioni a **32** bit della CPU, e riassembliamo gli esempi precedenti, possiamo constatare attraverso i **listing files** che questa volta l'assembler al posto del codice di "ingresso" ha inserito l'istruzione **ENTER**, mentre al posto del codice di "uscita" ha inserito l'istruzione **LEAVE**; queste due istruzioni sono disponibili solo con le CPU **80186** e superiori e hanno proprio lo scopo di rendere automatica la generazione del codice relativo allo **stack frame** di una procedura dotata di parametri e variabili locali.

L'istruzione **ENTER** ha il codice macchina **C8h** e richiede due operandi; il primo operando e' di tipo **imm16** mentre il secondo e' di tipo **imm8**. Il primo operando indica il numero di byte da sottrarre a **SP/ESP** per l'allocazione nello stack della memoria da destinare alle variabili locali; il secondo operando e' un numero compreso tra **0** e **31**, che indica il "livello di innesto", cioe' l'eventuale presenza di **stack frames** precedenti da innestare nel nuovo **stack frame** creato da **ENTER**. In questo modo e' possibile preservare **stack frames** precedentemente creati da una procedura; nel caso generale il secondo parametro vale **0**.
Applicando questi concetti alla procedura **TestProc** di Figura 4 (**6** byte di variabili locali), tutto il codice di "ingresso" viene sostituito dall'istruzione:

```
ENTER 6, 0
```

Quando la CPU incontra questa istruzione, salva **BP** nello stack, copia **SP** in **BP** e sottrae **6** byte a **SP**; in modalita' protetta a **32** bit la CPU lavora sui registri **ESP**, **EBP**

L'istruzione **LEAVE** ha il codice macchina **C9h** e non richiede nessun operando; il compito di

questa istruzione e' quello di ripristinare i registri **BP/EBP**, **SP/ESP** precedentemente modificati da **ENTER**.
Nel caso della procedura **TestProc** di Figura 4, tutto il codice di "uscita" viene sostituito dall'istruzione:

```
LEAVE
```

Quando la CPU incontra questa istruzione, copia **BP** in **SP** e ripristina **BP**; in modalita' protetta a **32** bit la CPU lavora sui registri **ESP**, **EBP**

28.13 Considerazioni finali

Appare evidente che gli strumenti avanzati che sono stati illustrati in questo capitolo, hanno principalmente lo scopo di semplificare il lavoro di interfacciamento tra l'**Assembly** e i linguaggi di alto livello; naturalmente nessuno ci impedisce di sviluppare programmi interamente scritti in **Assembly**, che fanno uso delle convenzioni adottate dai compilatori . I puristi dell'**Assembly** ritengono pero' che non avrebbe molto senso utilizzare in questo modo un linguaggio di basso livello; in effetti, se si ha bisogno di tutte queste caratteristiche avanzate, si fa molto prima a passare ad un linguaggio di alto livello.
Si tenga anche presente che quando si deve scrivere un modulo **Assembly** particolarmente complesso, le varie direttive semplificate illustrate in questo capitolo possono rivelarsi del tutto inadeguate; in una situazione del genere gli stessi manuali di **TASM** e **MASM** consigliano in particolare di utilizzare la sintassi classica per definire i segmenti di un programma.
In ogni caso, tutto dipende dai gusti personali; l'importante e' capire che per poter utilizzare tutte queste caratteristiche avanzate, e' necessaria una conoscenza piuttosto solida del linguaggio **Assembly** e delle convenzioni illustrate in questo capitolo.

Capitolo 30 - Interfaccia tra C e Assembly

In questo capitolo vengono descritte le convenzioni che permettono di interfacciare il linguaggio **C** con il linguaggio **Assembly**; naturalmente si da per scontato che chi legge abbia una adeguata conoscenza della programmazione in **C**.

Il linguaggio **C** e' stato creato nel **1972** da **Dennis Ritchie** presso i laboratori **Bell Telephone** della **AT&T**; l'implementazione del **C** venne effettuata su un computer **Digital Equipment PDP-11** gestito da un sistema operativo **UNIX**. Sino ad allora erano stati realizzati diversi linguaggi di programmazione come il **BASIC**, il **Pascal**, il **FORTRAN**, il **COBOL**, etc; tutti questi linguaggi si rivolgevano ad utenti accomunati dalla necessita' di programmare un computer senza avere la minima conoscenza degli aspetti legati all'architettura hardware degli elaboratori elettronici. Il **BASIC** ed il **Pascal** ad esempio si proponevano di insegnare l'arte della programmazione ai principianti; il **FORTRAN** permetteva di risolvere problemi matematici con il computer, mentre il **COBOL** veniva usato per scrivere programmi destinati al settore finanziario e alla gestione aziendale.
L'idea di **Dennis Ritchie** fu quella di creare un linguaggio di programmazione di alto livello e di uso generale (**general purpose**), rivolto pero' ad una utenza formata da esperti del settore dell'informatica e dell'hardware; a tale proposito **Ritchie** prese il linguaggio **Assembly** e gli aggiunse i costrutti sintattici tipici dei linguaggi di alto livello. In questo modo ottenne un linguaggio che permetteva di programmare un computer senza gli impedimenti e le limitazioni tipiche dei linguaggi destinati ai non esperti; in pratica, utilizzando il **C** il programmatore aveva la possibilita' di fare tutto cio' che era consentito dalla logica del computer.
Queste carattersitiche fecero guadagnare al **C** il titolo di linguaggio di programmazione per eccellenza, e di linguaggio ufficiale del mondo **UNIX**; nel corso della sua storia il **C** e' stato utilizzato per la realizzazione di sofisticati software che richiedevano le piu' elevate doti di potenza ed efficienza. In particolare il **C** e' diventato il linguaggio preferito dai progettisti di **SO**; con il **C** sono stati infatti realizzati i piu' famosi e diffusi **SO** come il **DOS**, **Windows**, **OS/2**, **UNIX**, **Linux** e molti altri. Un'altro settore dominato dal **C** e' quello della progettazione dei compilatori e degli interpreti; con il **C** sono stati realizzati numerosi compilatori ed interpreti per linguaggi come il **C++**, **Visual Basic**, **Java** e persino assemblatori come **MASM** e **TASM**. A tutto cio' bisogna anche aggiungere che nel **1983** un apposito comitato dell'**ANSI** (American National Standard Institute) ha definito lo standard ufficiale per il linguaggio **C** (**ANSI C**); l'importantissima conseguenza di tutto cio' sta nel fatto che il codice sorgente di un programma **ANSI C** puo' essere ricompilato senza nessuna modifica su qualsiasi piattaforma hardware dotata di un compilatore **ANSI C**.

Come e' stato detto in precedenza, la caratteristica rivoluzionaria del **C** sta nel fatto che questo linguaggio puo' essere definito un vero e proprio **Assembly** di alto livello che permette di fare tutto cio' che e' consentito dalla logica del computer; per chiarire questo concetto vediamo un semplice esempio pratico. Supponiamo che nel blocco dati di un programma **C** siano state dichiarate le seguenti variabili:

```
int      v[] = {2, 1, 4, 3, 8, 5};  /* vettore di 6 interi */
char     c1 = 12;                   /* intero a 8 bit */
char     c2 = 20;                   /* intero a 8 bit */
int      i1 = 12500;                /* intero a 16 bit */
int      i2 = 25300;                /* intero a 16 bit */
```

Un programmatore **Assembly** vedendo questo blocco dati capisce subito che la prima **WORD** del vettore **v** si trova a **+0** byte di distanza dall'inizio di **v**, la seconda **WORD** si trova a **+2** byte di distanza, la terza a **+4** byte di distanza e cosi' via sino alla sesta **WORD** che si trova a **+10** byte di distanza dall'offset iniziale assegnato a **v**; di conseguenza, il **BYTE** chiamato **c1** si trova a **+12** byte di distanza dall'inizio di **v**, il **BYTE** chiamato **c2** si trova a **+13** byte di distanza, la **WORD** chiamata **i1** si trova a **+14** byte di distanza e la **WORD** chiamata **i2** si trova a **+16** byte di distanza. Tutto cio' significa quindi che in **Assembly** possiamo scrivere ad esempio:

```
mov      al, byte ptr v[12]   ; carica c1 in AL
mov      ah, byte ptr v[13]   ; carica c2 in AH
mov      bx, v[14]            ; carica i1 in BX
mov      cx, v[16]            ; carica i2 in CX
```

Come sappiamo infatti, in **Assembly** la scrittura **v[12]** rappresenta il contenuto della locazione di memoria che si trova a **+12** byte di distanza dall'offset iniziale assegnato a **v** (al posto di **v[12]** si puo' anche scrivere **[v+12]**); di conseguenza l'istruzione:

```
mov al, byte ptr v[12]
```

carica in **AL** il contenuto a **8** bit della locazione di memoria che si trova a **+12** byte di distanza dall'inizio del vettore **v** (il modificatore **byte ptr** e' necessario in quanto **v** e' stato definito come vettore di **WORD** e non di **BYTE**).
Come si puo' notare, l'**Assembly** si fida del programmatore e non effettua quindi nessuna verifica sulla correttezza dell'indice che abbiamo assegnato al vettore **v**; in sostanza l'**Assembly** presuppone che il programmatore sappia esattamente cio' che sta facendo. Questa e' la stessa filosofia adottata nel linguaggio **C**; se vogliamo quindi visualizzare sullo schermo il contenuto delle variabili illustrate in precedenza, possiamo scrivere il seguente codice **C**:

```
for (i = 0; i < 6; i ++)
   printf("%d\n", v[i]);              /* visualizza v */

printf("%d\n", v[6] & 0x00ff);        /* visualizza c1 */
printf("%d\n", (v[6] >> 8) & 0x00ff); /* visualizza c2 */
printf("%d\n", v[7]);                 /* visualizza i1 */
printf("%d\n", v[8]);                 /* visualizza i2 */
```

Questo esempio ci offre l'occasione per sottolineare ancora una volta una differenza importantissima che esiste tra l'**Assembly** e i linguaggi di alto livello; in **Assembly**, se vogliamo accedere ad un elemento di un vettore, siamo tenuti a specificare lo spiazzamento in byte che possiede l'elemento all'interno del vettore stesso. Se ad esempio vogliamo accedere alla terza **WORD** del vettore **v** definito in precedenza, dobbiamo scrivere **v[4]**; infatti, la terza **WORD** di **v** si trova a **+4** byte di distanza dall'inizio dello stesso vettore **v**.
Nei linguaggi di alto livello invece tutti questi calcoli spettano al compilatore o all'interprete; di conseguenza il programmatore deve indicare tra parentesi quadre l'indice dell'elemento di un vettore e non lo spiazzamento. Nel linguaggio **C** ad esempio gli indici dei vettori partono da **0**, per cui se vogliamo accedere alla terza **WORD** di **v** dobbiamo scrivere **v[2]**; quando il compilatore incontra l'indice **2** lo moltiplica per la dimensione in byte (**2**) di ciascun elemento di **v** e ottiene lo spiazzamento **4**.

Le tecniche di indirizzamento illustrate in precedenza per il linguaggio **C**, vengono considerate sintatticamente illegali da molti altri linguaggi di programmazione; questo accade in particolare con quei linguaggi che essendo rivolti ai principianti, impongono precauzionalmente rigide regole sintattiche. Supponiamo ad esempio che il blocco dati illustrato in precedenza appartenga ad un programma scritto in **Pascal**, e supponiamo anche che il vettore **v** sia stato definito come:

```
var v: array[1..6] of Integer;
```

Se ora proviamo a scrivere **v[7]**, otteniamo un messaggio di errore del compilatore **Pascal**; questo messaggio ci informa che stiamo utilizzando un indice fuori limite per il vettore **v**. In ogni caso si tenga presente che spesso queste regole sono facilmente aggirabili; osservando ad esempio che **i1** si trova a **8 WORD** di distanza dall'inizio di **v**, possiamo definire una variabile intera **i** alla quale assegnamo il valore **8** in modo da poter scrivere il seguente codice **Pascal**:

```
WriteLn('i1 = ', v[i]);
```

Un qualsiasi compilatore, interprete o assemblatore non puo' effettuare in questo caso nessuna verifica sul contenuto dell'indice del vettore; infatti l'indice **i** e' una variabile e non una costante, per cui il suo contenuto sara' noto (alla CPU) solo in fase di esecuzione del programma.

Un'altro esempio pratico e' rappresentato dal fatto che se in **C** definiamo una variabile **x** di tipo **float** (numero reale a **32** bit) e una variabile **i3** di tipo **long** (numero intero con segno a **32** bit), possiamo scrivere allora il seguente assegnamento:

```
i3 = x;
```

Dal punto di vista della CPU questa istruzione e' perfettamente legale in quanto consiste nel copiare il contenuto a **32** bit di **x** (in formato **IEEE** standard), nel contenuto a **32** bit di **i3**; di conseguenza, anche i compilatori **C** considerano legale questo assegnamento. Un compilatore **Pascal** invece incontrando la precedente istruzione produce un messaggio di errore; questo perche' il **Pascal** impone rigide regole sulla compatibilita' tra diversi tipi di dati, e quindi non permette di copiare una variabile reale in una variabile intera anche se in memoria entrambe le variabili rappresentano locazioni da **32** bit.

Come e' stato detto nei precedenti capitoli, un qualunque assemblatore, compilatore o intrerprete (**traduttore di linguaggio**), e' predisposto per poter funzionare su una ben precisa piattaforma hardware; in questo modo e' possibile generare programmi capaci di sfruttare la particolare architettura del computer sul quale si sta lavorando. Tutte le considerazioni che verranno esposte nel seguito del capitolo, si riferiscono come al solito alle piattaforme hardware basate su CPU **Intel 80x86** e compatibili; come gia' sappiamo, questa piattaforma prevede due modalita' operative fondamentali che sono la **modalita' reale** a **16** bit e la **modalita' protetta** che a partire dalle CPU **80386**, permette di sfruttare l'architettura a **32** bit del computer.

In seguito, quando si parla di ambiente operativo a **16** bit, ci si riferisce alla modalita' operativa reale delle CPU **Intel 80x86** e compatibili; come gia' sappiamo, un traduttore di linguaggio che lavora in un ambiente operativo a **16** bit, puo' essere in grado di generare programmi capaci di sfruttare l'eventuale presenza di CPU a **32** bit. In questo caso si ottengono programmi che utilizzano il set di istruzioni a **32** bit senza uscire dalla modalita' reale; in ogni caso, per garantire la compatibilita' con le CPU a **16** bit, gli indirizzi di memoria restano sempre o di tipo **NEAR** (formati da un **offset** a **16** bit), o di tipo **FAR** (formati da una coppia **seg:offset** a **16+16** bit).

Quando si parla di ambiente operativo a **32** bit, ci si riferisce alla modalita' operativa protetta delle CPU **Intel 80x86** e compatibili; un traduttore di linguaggio che lavora in un ambiente operativo a **32** bit, genera programmi destinati a girare in modalita' protetta a **32** bit, capaci di sfruttare gli indirizzamenti di tipo **NEAR** a **32** bit (formati da un **offset** a **32** bit).

Questo capitolo tratta l'interfaccia tra **C** e **Assembly** nell'ambiente operativo a **16** bit offerto dal **SO DOS**; per lo sviluppo dei programmi di esempio presentati nel seguito e' necessario quindi munirsi di un compilatore **C** a **16** bit come il **Microsoft C**, il **Borland TurboC**, etc. Gli aspetti relativi alla modalita' protetta delle CPU vengono illustrati nella sezione **Modalita' Protetta**; gli aspetti relativi alla interfaccia tra **C** e **Assembly** nell'ambiente operativo a **32** bit di **Windows** vengono illustrati nella sezione **Win32 Assembly**.

29.1 Tipi di dati del C

Per poter interfacciare correttamente un linguaggio di alto livello con l'**Assembly**, e' necessario conoscere tutte le informazioni relative ai tipi di dati forniti dal linguaggio stesso; queste informazioni assumono una importanza vitale nell'interscambio di dati tra due o piu' moduli di un programma e in particolare nella fase di gestione dello **stack frame** di una procedura.
Un compilatore **C** destinato a lavorare in un ambiente operativo a **16** bit, fornisce una serie di tipi di dati interi aventi le seguenti carattersitiche:

Nome	Tipo	Val. Min.	Val. Max.
signed char	intero con segno a 8 bit	-128	+127
unsigned char	intero senza segno a 8 bit	0	+255
signed short int	intero con segno a 16 bit	-32768	+32767
unsigned short int	intero senza segno a 16 bit	0	+65535
signed int	intero con segno a 16 bit	-32768	+32767
unsigned int	intero senza segno a 16 bit	0	+65535
signed long int	intero con segno a 32 bit	-2147483648	+2147483647
unsigned long int	intero senza segno a 32 bit	0	+4294967295

Nel linguaggio **C** il tipo **int** rappresenta il tipo di dato intero di riferimento, e in base a quanto previsto dallo standard **ANSI C**, la sua ampiezza in bit deve rispecchiare il numero di bit che caratterizza l'ambiente operativo a cui e' destinato il compilatore. Proprio per questo motivo i compilatori destinati agli ambienti operativi a **16** bit forniscono il tipo di dato **int** con ampiezza pari a **16** bit; lo standard **ANSI C** prevede inoltre che il tipo **short int** abbia una ampiezza in bit non superiore a quella di un **int**, e che il tipo **long int** abbia una ampiezza in bit non inferiore a quella di un **int**.
Il **C** fornisce anche il tipo **enum** che permette di definire enumerazioni di dati di tipo **signed int**; questi dati hanno quindi la stessa ampiezza in bit dei **signed int**.

I tipi di dati in virgola mobile (**floating point**), sono indipendenti dall'architettura del computer in quanto il loro formato standard viene stabilito come sappiamo dall'**IEEE** (Institute of Electrical and Electronics Engineers); i compilatori **C** che supportano lo standard **IEEE** forniscono i seguenti tre tipi di dati in virgola mobile:

Nome	Tipo	Val. Min.	Val. Max.	Precisione
float	reale con segno a 32 bit	3.4×10^{-38}	3.4×10^{38}	6-7 cifre
double	reale con segno a 64 bit	1.7×10^{-308}	1.7×10^{308}	15-16 cifre
long double	reale con segno a 80 bit	3.4×10^{-4932}	1.1×10^{4932}	19 cifre

Gli estremi **Min.** e **Max.** si riferiscono ai numeri reali positivi; per ottenere gli estremi negativi basta mettere il segno meno davanti agli estremi positivi. Il campo **Precisione** si riferisce al numero di cifre esatte dopo la virgola; si tenga presente che tanto maggiore e' la parte intera del numero reale, tanto minore sara' il numero di cifre esatte dopo la virgola.

Riassumendo possiamo dire che tutti questi aspetti dipendono dall'architettura del computer che si sta utilizzando; per convenzione, i limiti minimo e massimo che assumono i tipi interi e in virgola mobile, vengono specificati nei due header **limits.h** e **float.h** della libreria standard del **C**. Il seguente programma **C** permette di verificare in pratica le considerazioni esposte in precedenza:

```
/* file datatyp.c */

#include < stdio.h >

/* variabili globali */

signed char          sc1 = 127;
unsigned char        uc1 = 240;
signed short int     ssi1 = 12000;
unsigned short int   usi1 = 40000U;
signed int           si1 = 12000;
unsigned int         ui1 = 40000U;
signed long int      sli1 = 2000000000L;
unsigned long int    uli1 = 4000000000UL;

float                fl1 = -2.71828181543234567893;
double               db1 = +3.23658976584456723456;
long double          ld1 = +0.87350125734897445787L;

/* entry point */

int main(void)
{
   printf("%10d, %d\n", sc1, sizeof(sc1));
   printf("%10d, %d\n", uc1, sizeof(uc1));
   printf("%10hd, %d\n", ssi1, sizeof(ssi1));
   printf("%10hu, %d\n", usi1, sizeof(usi1));
   printf("%10d, %d\n", si1, sizeof(si1));
   printf("%10u, %d\n", ui1, sizeof(ui1));
   printf("%10ld, %d\n", sli1, sizeof(sli1));
   printf("%10lu, %d\n", uli1, sizeof(uli1));
   printf("\n");
   printf("%25.20f, %2d\n", fl1, sizeof(fl1));
   printf("%25.20f, %2d\n", db1, sizeof(db1));
   printf("%25.20Lf, %2d\n", ld1, sizeof(ld1));

   return 0;
}
```

Compilando questo programma e' possibile verificare le diverse ampiezze in bit assunte dai tipi interi; in relazione invece ai numeri reali e' possibile verificare il differente numero di cifre decimali esatte garantito dai tre tipi **float**, **double** e **long double**.

29.2 Convenzioni per i compilatori C

Segmenti di programma

In relazione ai segmenti di programma, i compilatori **C** seguono completamente le convenzioni illustrate nei precedenti capitoli; si puo' dire anzi che queste convenzioni sono state concepite in particolare proprio per i compilatori **C**.
In ambiente operativo a **16** bit su piattaforme hardware basate su CPU **Intel 80x86** e compatibili, i compilatori **C** supportano la seguente segmentazione standard:

Segmento dati inizializzati **NEAR**: _DATA SEGMENT WORD PUBLIC USE16 'DATA' con direttiva semplificata **.DATA**
Segmento dati non inizializzati **NEAR**: _BSS SEGMENT WORD PUBLIC USE16 'BSS' con direttiva semplificata **.DATA?**
Segmento dati costanti **NEAR**: _CONST SEGMENT WORD PUBLIC USE16 'CONST' con direttiva semplificata **.CONST**
Segmento di stack: _STACK SEGMENT PARA STACK USE16 'STACK' con direttiva semplificata **.STACK [size]**
Segmento dati inizializzati **FAR**: FAR_DATA SEGMENT PARA PRIVATE USE16 'FAR_DATA' con direttiva semplificata **.FARDATA [name]**
Segmento dati non inizializzati **FAR**: FAR_BSS SEGMENT PARA PRIVATE USE16 'FAR_BSS' con direttiva semplificata **.FARDATA? [name]**
Segmento di codice: name_TEXT SEGMENT WORD PUBLIC USE16 'CODE' con direttiva semplificata **.CODE [name]**

I vari blocchi di dati **NEAR** vengono raggruppati dai compilatori **C** con la direttiva:

```
DGROUP GROUP _DATA, _CONST, _STACK, _BSS
```

Il registro **DS** viene automaticamente inizializzato dal compilatore **C** con **DGROUP**, mentre **CS** viene inizializzato con il blocco **_TEXT** contenente l'**entry point** del programma; in genere, nei modelli di memoria che prevedono due o piu' blocchi di dati, il blocco **_STACK** viene tenuto separato da **DGROUP**. Il programmatore puo' anche richiedere esplicitamente al compilatore **C** di tenere separato il blocco **_STACK** dal gruppo **DGROUP**; nel caso ad esempio del compilatore **Borland C++ 3.1**, dall'interno dell'editor integrato bisogna selezionare il menu **Options - Compiler - Code generation** e disattivare l'opzione **Assume SS Equals DS**. In ogni caso, il registro **SS** viene inizializzato con il segmento o il gruppo in cui si trova il blocco **_STACK**.
Come e' stato detto nel precedente capitolo, quando si interfaccia l'**Assembly** con i linguaggi di alto livello, e' vivamente consigliabile l'uso delle direttive semplificate per i segmenti; in questo modo si delega all'assembler il compito di espandere queste direttive nel modo piu' adeguato ai parametri indicati nella direttiva **.MODEL**. Abbiamo anche visto che in presenza delle direttive semplificate per i segmenti, l'assembler crea automaticamente il gruppo **DGROUP**; definendo invece i segmenti di programma con la sintassi illustrata nella tabella precedente, tutti questi aspetti sono a carico del programmatore che in questo caso deve sapere esattamente come comportarsi.

Stack frame

In relazione allo **stack frame** bisogna ricordare innanzi tutto che in **C** le procedure vengono chiamate **funzioni**; nei precedenti capitoli abbiamo visto che gli eventuali argomenti richiesti da una funzione **C** vengono inseriti nello stack a partire dall'ultimo (destra sinistra). In questo modo, all'interno dello stack i parametri di una funzione si trovano posizionati nello stesso ordine indicato dal prototipo della funzione stessa; il compito di ripulire lo stack dagli argomenti spetta come sappiamo al caller. Per accedere ai parametri di una funzione, i compilatori **C** a **16** bit si servono di **BP**; abbiamo visto che i parametri si trovano a spiazzamenti positivi da **BP**, mentre le eventuali variabili locali si trovano a spiazzamenti negativi da **BP**.

Valori di ritorno

In relazione infine alle locazioni utilizzate per contenere gli eventuali valori di ritorno delle funzioni, valgono tutte le convenzioni illustrate nel **Capitolo 24**; in ambiente operativo a **16** bit su piattaforme hardware basate su CPU **Intel 80x86** e compatibili, le convenzioni per i valori di ritorno delle funzioni **C** prevedono quanto segue:

Gli interi a **8** bit vengono restituiti negli **8** bit meno significativi (**AL**) di **AX**.
Gli interi a **16** bit (compresi gli offset **NEAR**) vengono restituiti in **AX**.
Gli interi a **32** bit vengono restituiti nella coppia **DX:AX**, con **DX** che in base alla notazione **little-endian** contiene la **WORD** piu' significativa.
Gli indirizzi **FAR** a **16+16** bit vengono restituiti sempre nella coppia **DX:AX**, con **DX** che contiene la componente **seg**, e **AX** la componente **offset**.
Tutti i valori in virgola mobile **IEEE** di tipo **float** a **32** bit, **double** a **64** bit e **long double** a **80** bit, vengono restituiti nel registro **ST(0)** della **FPU**; il registro **ST(0)** e' la cima dello stack (**TOS**) del coprocessore matematico.

Sempre in relazione ai valori di ritorno, bisogna ricordare che il **C** permette di definire funzioni che possono restituire intere strutture, non solo per indirizzo, ma anche per valore; ovviamente, il caso di una funzione che restituisce una struttura per indirizzo e' molto semplice. Abbiamo appena visto infatti che gli indirizzi **NEAR** vengono restituiti in **AX**, mentre gli indirizzi **FAR** vengono restituiti in **DX:AX**; il caso interessante riguarda invece il procedimento utilizzato dai compilatori **C** per permettere ad una funzione di restituire una struttura per valore. Se la struttura occupa complessivamente **1**, **2** o **4** byte, viene restituita rispettivamente nei registri **AL**, **AX**, **DX:AX**, mentre se la struttura occupa complessivamente **3** byte, oppure piu' di **4** byte, viene restituita al caller attraverso un metodo piu' complesso; questo aspetto verra' illustrato nel seguito del capitolo attraverso un apposito esempio pratico.

29.3 Le classi di memoria del C

Nella terminologia dei linguaggi di alto livello, con l'espressione **classe di memoria** si indica l'insieme degli aspetti legati alla visibilita' e alla durata di un identificatore; questi aspetti coinvolgono naturalmente il metodo attraverso il quale l'identificatore e' stato creato. Come accade nella maggior parte dei linguaggi, anche in **C** esistono due categorie fondamentali di identificatori, e cioe' gli identificatori **locali** e **globali**; in generale, gli identificatori locali del **C** sono quelli definiti all'intero di un blocco funzionale delimitato da una coppia **{ }**, mentre gli identificatori globali sono quelli esterni a qualsiasi blocco **{ }**.

A differenza di quanto accade in **Pascal**, il **C** non permette di definire funzioni innestate all'interno di altre funzioni; possiamo dire quindi che in **C** gli identificatori delle funzioni sono tutti globali, e quindi le funzioni esistono per tutta la fase di esecuzione e sono visibili in tutto il programma. In assenza di altre indicazioni da parte del programmatore, una funzione **C** definita in un modulo e' visibile anche in altri moduli del programma; se in un modulo **A** definiamo una funzione **func1** con prototipo:

```
int func1(char, char);
```

allora in un modulo **B** possiamo rendere visibile **func1** attraverso la dichiarazione:

```
extern int func1(char, char);
```

Per fare in modo che **func1** non sia visibile all'esterno del modulo **A**, dobbiamo utilizzare la parola chiave **static** e dichiarare in **A** il prototipo:

```
static int func1(char, char);
```

Le etichette del **C** possono essere definite solo all'interno di una funzione; la loro visibilita' quindi e' limitata alla funzione di appartenenza, e il loro identificatore puo' essere ridefinito in altri punti del programma.

Le variabili del **C** possono essere sia globali che locali; in **C** le variabili globali vengono anche chiamate **statiche**, mentre le variabili locali vengono anche chiamate **automatiche**. Una variabile e' globale quando viene definita all'esterno di qualsiasi blocco funzionale **{ }**; in questo caso il compilatore **C** inserisce la variabile globale nel blocco dati del programma. Una variabile globale quindi esiste per tutta la fase di esecuzione, ed e' visibile in tutto il programma. Le variabili globali possono essere inizializzate solo con valori costanti; in assenza di inizializzazione, lo standard **ANSI C** prevede che il compilatore assegni ad una variabile globale il valore iniziale **0**.
In assenza di altre indicazioni da parte del programmatore, una variabile globale del **C** definita in un modulo e' visibile anche in altri moduli del programma; se in un modulo **A** definiamo una variabile globale:

```
float pi_greco = 3.14;
```

allora in un modulo **B** possiamo rendere visibile **pi_greco** attraverso la dichiarazione:

```
extern float pi_greco;
```

Per fare in modo che **pi_greco** non sia visibile all'esterno del modulo **A**, dobbiamo utilizzare la parola chiave **static** e riscrivere nel modulo **A** la definizione:

```
static float pi_greco = 3.14;
```

Una variabile del **C** e' locale (o automatica) quando viene definita all'interno di un qualsiasi blocco **{ }**; la visibilita' di una variabile locale e' limitata al blocco **{ }** di appartenenza, e quindi il suo identificatore puo' essere ridefinito altrove. Una variabile locale viene creata nello stack quando si entra nel blocco **{ }** di appartenenza, e viene poi distrutta quando si esce dal blocco stesso; per fare in modo che una variabile locale non venga distrutta, bisogna utilizzare ancora la parola chiave **static**. Se la precedente variabile **pi_greco** viene definita all'interno di un blocco **{ }**, possiamo scrivere:

```
static float pi_greco = 3.14;
```

In questa seconda versione la parola chiave **static** dice al compilatore **C** che **pi_greco** non deve

essere distrutta all'uscita dal blocco **{ }** di appartenenza; il trucco adottato dal compilatore **C** consiste nel creare questo tipo di variabili nel blocco dati del programma e non nello stack. E' perfettamente lecito quindi il caso di una funzione che restituisce al caller l'indirizzo di una sua variabile locale **static**; questa variabile infatti essendo stata definita nel blocco dati del programma, continua ad esistere anche al di fuori della funzione che l'ha creata.
Questo stesso discorso vale anche se all'interno di una funzione definiamo una variabile locale del tipo:

```
char *loc_string = "Stringa locale";
```

Quando il compilatore incontra questa definizione, crea la stringa **"Stringa locale"** nel blocco dati del programma, poi crea nello stack una variabile locale **loc_string** di tipo puntatore a **char** e le assegna l'indirizzo iniziale della stringa; al termine della funzione, la variabile locale **loc_string** viene distrutta, mentre la stringa **"Stringa locale"** continua ad esistere. In sostanza, anche in assenza della parola chiave **static**, le stringhe definite con il precedente metodo all'interno di un blocco **{ }**, vengono ugualmente sistemate nel blocco dati del programma; e' quindi lecito scrivere una funzione **C** che restituisce al caller l'indirizzo contenuto nella precedente variabile puntatore **loc_string**.
Le variabili locali del **C** possono essere inizializzate sia con valori costanti che con il contenuto di altre variabili; in assenza di inizializzazione il contenuto di una variabile locale e' casuale ("sporco").

29.4 Gestione degli indirizzamenti in C

Nel linguaggio **C** un puntatore e' una variabile contenente un valore intero che rappresenta un indirizzo di memoria; negli ambienti operativi a **16** bit un indirizzo di memoria puo' essere o di tipo **NEAR** (formato da un **offset** a **16** bit), o di tipo **FAR** (formato da una coppia **seg:offset** a **16+16** bit). Come gia' sappiamo, l'indirizzo di una locazione di memoria da **8** bit, **16** bit, **32** bit, etc, coincide con l'indirizzo del byte meno significativo della locazione stessa; un puntatore contiene quindi l'indirizzo **NEAR** o **FAR** del byte meno significativo della locazione di memoria a cui sta puntando.
Analizziamo in pratica questi concetti premettendo che nel seguito, quando si parla genericamente di compilatore, ci si riferisce al processo di traduzione del codice **C** in codice macchina; supponiamo di avere il seguente blocco dati di un programma **C**:

```
char     c1 = 12;
int      i1 = 1350;
long     l1 = 186900;
```

Quando un compilatore **C** incontra questo blocco dati:

***** assegna a **c1** una locazione di memoria da **8** bit, e carica in questa locazione il valore **12**;
***** assegna a **i1** una locazione di memoria da **16** bit, e carica in questa locazione il valore **1350**;
***** assegna a **l1** una locazione di memoria da **32** bit, e carica in questa locazione il valore **186900**.

Supponendo di lavorare con indirizzi di tipo **NEAR**, in presenza della definizione:

```
char *pc1 = &c1;
```

il compilatore assegna a **pc1** una locazione di memoria da **16** bit, e carica in questa locazione l'**offset** iniziale di **c1**.
In presenza della definizione:

```
int *pi1 = &i1;
```

il compilatore assegna a **pi1** una locazione di memoria da **16** bit, e carica in questa locazione l'**offset** iniziale di **i1**.

In presenza della definizione:
`long *pl1 = &l1;`
il compilatore assegna a **pl1** una locazione di memoria da **16** bit, e carica in questa locazione l'**offset** iniziale di **l1**.

Se invece stiamo lavorando con indirizzi di tipo **FAR**, in presenza della definizione:
`char far *pc1 = &c1;`
il compilatore assegna a **pc1** una locazione di memoria da **16+16** bit, e carica in questa locazione la coppia **seg:offset** iniziale di **c1**.
In presenza della definizione:
`int far *pi1 = &i1;`
il compilatore assegna a **pi1** una locazione di memoria da **16+16** bit, e carica in questa locazione la coppia **seg:offset** iniziale di **i1**.
In presenza della definizione:
`long far *pl1 = &l1;`
il compilatore assegna a **pl1** una locazione di memoria da **16+16** bit, e carica in questa locazione la coppia **seg:offset** iniziale di **l1**.

A questo punto, se all'interno di una funzione **C** scriviamo:
`char c2 = *pc1;`
il compilatore accede all'indirizzo puntato da **pc1**, legge un blocco da **8** bit che parte da quell'indirizzo, e lo copia nella locazione a **8** bit in cui si trova **c2**.
Se scriviamo:
`int i2 = *pi1;`
il compilatore accede all'indirizzo puntato da **pi1**, legge un blocco da **16** bit che parte da quell'indirizzo, e lo copia nella locazione a **16** bit in cui si trova **i2**.
Se scriviamo:
`long l2 = *pl1;`
il compilatore accede all'indirizzo puntato da **pl1**, legge un blocco da **32** bit che parte da quell'indirizzo, e lo copia nella locazione a **32** bit in cui si trova **l2**.

Supponiamo ora di avere una funzione scritta in **Assembly** che presenta il seguente prototipo **C**:
`void func_asm(char c, int i, long l);`
Effettuiamo ora da un modulo scritto in **C**, la chiamata di **func_asm** con gli argomenti **c1**, **i1** e **l1** definiti in precedenza; la chiamata assume il seguente aspetto:
`func_asm(c1, i1, l1);`
In questo caso abbiamo a che fare con il passaggio per valore dei tre argomenti **c1**, **i1** e **l1**, per cui vengono create nello stack le copie dei valori di questi tre argomenti; queste tre copie vengono chiamate simbolicamente **c**, **i** e **l** e rappresentano i tre parametri della funzione **func_asm**.
All'interno di **func_asm** il parametro **c** rappresenta un generico valore intero **12** a **8** bit, il parametro **i** rappresenta un generico valore intero **1350** a **16** bit, mentre il parametro **l** rappresenta un generico valore intero **186900** a **32** bit; in base a queste considerazioni, nella funzione **func_asm** possiamo scrivere quindi istruzioni del tipo:

```
xor      eax, eax    ; eax = 0
mov      al, c       ; al = c
add      ax, i       ; ax = i + c
mov      ebx, l      ; ebx = l
sub      ebx, eax    ; ebx = l - (i + c)
```

E' importante notare che i vari operandi che fanno parte di una istruzione, devono avere dimensioni in bit compatibili con quelle previste dall'istruzione stessa; non avrebbe senso quindi utilizzare **MOV** per caricare **c** in **AX**, o utilizzare **ADD** per sommare **EBX** con **i**.
Grazie al passaggio per valore, qualsiasi modifica apportata ai parametri **c**, **i** e **l**, non ha nessuna ripercussione sugli argomenti **c1**, **i1** e **l1**; infatti, **c**, **i** e **l** si trovano nel blocco stack, mentre **c1**, **i1** e **l1** si trovano nel blocco dati del programma.

Vediamo invece come ci si deve comportare quando il prototipo della funzione **func_asm** e:
```
void func_asm(char *pc, int *pi, long *pl);
```
Utilizzando le variabili definite in precedenza, la chiamata di questa funzione diventa:
```
func_asm(&c1, &i1, &l1);
```
oppure:
```
func_asm(pc1, pi1, pl1);
```
Nel primo caso stiamo passando direttamente gli indirizzi degli argomenti **c1**, **i1** e **l1**, per cui si parla di passaggio diretto per indirizzo; nel secondo caso stiamo passando indirettamente (tramite cioe' i puntatori) gli indirizzi degli argomenti **c1**, **i1** e **l1**, per cui si parla di passaggio indiretto per indirizzo.
In entrambi i casi, la funzione **func_asm** riceve tre valori interi a **16** bit chiamati simbolicamente **pc**, **pi** e **pl**, che rappresentano i tre indirizzi **NEAR** di **c1**, **i1** e **l1**; attraverso questi tre indirizzi, la funzione **func_asm** puo' accedere alle relative locazioni di memoria modificandone il contenuto originale.
Tenendo conto del significato dei parametri **pc**, **pi** e **pl**, all'interno di **func_asm** possiamo scrivere istruzioni del tipo:

```
mov      al, 3              ; al = 3
mov      si, pc             ; si = pc = &c1
add      [si], al           ; c1 = 12 + 3 = 15
mov      di, pi             ; di = pi = &i1
sub      word ptr [di], 4   ; i1 = 1350 - 4 = 1346
mov      bx, pl             ; bx = pl = &l1
shr      dword ptr [bx], 2 ; l1 = 186900 / 4 = 46725
```

Questa volta abbiamo a che fare con indirizzi **NEAR** formati ciascuno da una componente **offset** a **16** bit; questi indirizzi vengono gestiti attraverso i tre registri puntatori **SI**, **DI** e **BX**. Accedendo alle locazioni di memoria relative a questi indirizzi, possiamo modificarne il contenuto originale; infatti, come si vede nell'esempio, tutte le modifiche apportate a **[SI]**, **[DI]** e **[BX]** si ripercuotono su **c1**, **i1** e **l1**.
Da queste considerazioni si intuisce subito che se il parametro **pc** (che abbiamo caricato in **SI**) e' l'indirizzo iniziale di un vettore di **char**, allora gli indirizzi dei vari elementi del vettore corrispondono a: **SI+0**, **SI+1**, **SI+2**, etc; di conseguenza, il contenuto dei vari elementi del vettore e' rappresentato da: **[SI+0]**, **[SI+1]**, **[SI+2]**, etc.
Se il parametro **pi** (che abbiamo caricato in **DI**) e' l'indirizzo iniziale di un vettore di **int**, allora gli indirizzi dei vari elementi del vettore corrispondono a: **DI+0**, **DI+2**, **DI+4**, etc; di conseguenza, il contenuto dei vari elementi del vettore e' rappresentato da: **[DI+0]**, **[DI+2]**, **[DI+4]**, etc.
Se il parametro **pl** (che abbiamo caricato in **BX**) e' l'indirizzo iniziale di un vettore di **long**, allora gli indirizzi dei vari elementi del vettore corrispondono a: **BX+0**, **BX+4**, **BX+8**, etc; di conseguenza, il contenuto dei vari elementi del vettore e' rappresentato da: **[BX+0]**, **[BX+4]**, **[BX+8]**, etc.

Nel caso di indirizzamenti di tipo **FAR** bisogna solo ricordarsi che in questo caso si devono gestire indirizzi da **16+16** bit; il prototipo della funzione **func_asm** diventa:
```
void func_asm(char far *pc, int far *pi, long far *pl);
```
In presenza della chiamata:
```
func_asm(&c1, &i1, &l1);
```
il compilatore **C** provvede automaticamente a passare le coppie **seg:offset** relative agli indirizzi dei vari argomenti; per poter effettuare invece la chiamata:
```
func_asm(pc1, pi1, pl1);
```
dobbiamo servirci delle definizioni **FAR** illustrate in precedenza per i tre puntatori **pc1**, **pc2** e **pc3**.
In entrambi i casi, la funzione **func_asm** riceve tre valori interi a **16+16** bit chiamati simbolicamente **pc**, **pi** e **pl**, che rappresentano i tre indirizzi **FAR** di **c1**, **i1** e **l1**; attraverso questi tre indirizzi, la funzione **func_asm** puo' accedere alle relative locazioni di memoria modificandone il contenuto originale.
Tenendo conto del significato dei parametri **pc**, **pi** e **pl**, all'interno di **func_asm** possiamo scrivere istruzioni del tipo:

```
mov     al, 3                 ; al = 3
les     si, pc                ; es:si = pc = &c1
add     es:[si], al           ; c1 = 12 + 3 = 15
lfs     di, pi                ; fs:di = pi = &i1
sub     word ptr fs:[di], 4   ; i1 = 1350 - 4 = 1346
lgs     bx, pl                ; gs:bx = pl = &l1
shr     dword ptr gs:[bx], 2  ; l1 = 186900 / 4 = 46725
```

Queste istruzioni presuppongono l'utilizzo del set di istruzioni a **32** bit in modo da poter disporre dei registri di segmento **FS** e **GS**; in caso contrario bisogna riuscire a destreggiarsi con **DS** e **ES**, ricordandosi poi di ripristinare il contenuto originale di **DS**.
Se **pc**, **pi** e **pl** rappresentano gli indirizzi di partenza di tre vettori rispettivamente di **char**, di **int** e di **long**, valgono le stesse considerazioni gia' esposte per il caso degli indirizzi **NEAR**; l'unica differenza sta nel fatto che questa volta dobbiamo gestire anche la componente **seg** di ogni elemento dei vettori, per cui ad esempio **ES:[SI+4]** rappresenta il contenuto del quinto **char** del vettore puntato da **pc**.

29.5 Protocollo di comunicazione tra C e Assembly

In base alle considerazioni appena svolte e in base alle cose dette nei precedenti capitoli, possiamo dedurre una serie di regole che permettono a due o piu' moduli **C** e **Assembly** di comunicare tra loro; l'insieme di queste regole definisce quindi il protocollo di comunicazione tra **C** e **Assembly**.

I moduli **C** e **Assembly** destinati a comunicare tra loro, devono specificare lo stesso modello di memoria; questo aspetto e' di vitale importanza in quanto garantisce il corretto scambio di informazioni tra i vari moduli. In alternativa e' possibile seguire una strada molto complessa e pericolosa, che consiste nello specificare esplicitamente il tipo di indirizzamento **NEAR** o **FAR** necessario per accedere alle variabili e alle procedure del programma.

Un modulo **C** che intende utilizzare degli identificatori definiti in un modulo **Assembly**, deve dichiarare questi identificatori attraverso il qualificatore **extern**; da parte sua il modulo **Assembly** contenente le definizioni degli identificatori da utilizzare nel modulo **C**, deve specificare le necessarie direttive **PUBLIC** relative agli identificatori stessi.

Un modulo **Assembly** che intende utilizzare degli identificatori definiti in un modulo **C**, deve dichiarare questi identificatori attraverso la direttiva **EXTRN**; da parte sua il modulo **C** contenente le definizioni degli identificatori da utilizzare nel modulo **Assembly**, deve provvedere a rendere esternamente visibili gli identificatori stessi. Tutti gli identificatori visibili anche all'esterno del modulo di appartenenza, vengono chiamati **identificatori con linkaggio esterno**; gli identificatori visibili invece solo all'interno del modulo di appartenenza, vengono chiamati **identificatori con linkaggio interno**.

In un programma **C**, tutti gli identificatori con linkaggio esterno devono essere preceduti dal carattere **'_'** (underscore); il carattere underscore e' associato al codice ASCII **95**. I compilatori **C** aggiungono automaticamente l'underscore all'inizio di ogni identificatore con linkaggio esterno presente nei vari moduli **C** di un programma; per non generare confusione, nei moduli **C** e' consigliabile quindi evitare di utilizzare nomi preceduti dall'underscore. Nel caso degli identificatori con linkaggio esterno definiti o dichiarati nei moduli **Assembly**, si possono presentare due casi:
1) se si sta programmando in stile **Assembly** classico, il compito di inserie tutti gli underscores spetta al programmatore;
2) se invece si utilizzano le carattersitiche avanzate di **MASM** e **TASM**, basta passare il parametro di linguaggio **C** alla direttiva **.MODEL** per delegare all'assembler il compito di inserire tutti gli underscores.

Tutte le procedure contenute nei moduli **Assembly** da linkare al **C**, devono preservare rigorosamente il contenuto di tutti i registri di segmento (in particolare **CS**, **DS** e **SS**) e dei registri **SP** e **BP**; se si utilizzano le carattersitiche avanzate di **MASM** e **TASM**, all'interno delle procedure dotate di **stack frame** il contenuto dei registri **SP** e **BP** viene automaticamente preservato dall'assembler. I registri **CS** e **SS** vengono gestiti direttamente dal **SO** e non necessitano quindi di modifiche da parte del programmatore; in rarissime circostanze si puo' avere la necessita' di modificare **SS** e **SP** per poter gestire uno stack personalizzato. I registri di segmento **ES**, **FS** e **GS** sono a nostra completa disposizione, mentre **DS** viene inizializzato con **DGROUP** dal compilatore **C**; se il programmatore ha la necessita' di modificare **DS** per poter referenziare altri segmenti di dati, deve preservarne il contenuto originale (che in genere e' **DGROUP**).
I compilatori **C** utilizzano anche i registri **SI** e **DI** per contenere le variabili intere con qualificatore **register**; molti compilatori **C** permettono di abilitare o meno questa possibilita'. Per non creare problemi al compilatore, si consiglia quindi di preservare sempre **SI**, **DI** nelle proprie procedure **Assembly** che utilizzano questi registri; generalmente, tutte le funzioni delle librerie fornite da un compilatore **C** preservano **SI** e **DI**.

Il **C** e' un linguaggio **case-sensitive**, e quindi i compilatori **C** distinguono tra maiuscole e minuscole; questo significa che identificatori come **varword1**, **VarWord1**, **VARWORD1**, etc, verranno considerati tutti distinti. In osservanza a queste regole, al momento di assemblare un modulo **Assembly** da linkare ad un modulo **C**, e' necessario richiedere un assemblaggio **case-sensitive**; se si sta utilizzando **TASM** il parametro da passare all'assembler e' **/ml**, mentre se si sta utilizzando **MASM** il parametro da passare all'assembler e' **/Cp**.

29.6 Esempi pratici

Analizziamo ora una serie di esempi pratici che ci permettono di verificare le considerazioni esposte in precedenza; come e' stato gia' detto, tutti gli esempi che seguono si riferiscono all'ambiente operativo a **16** bit offerto dal **SO DOS**.

Chiamate incrociate tra moduli C e moduli Assembly

Cominciamo con un programma formato dai due moduli **CMOD1.C** e **ASMMOD1.ASM**; questi due moduli si chiamano a vicenda. Il modulo **CMOD1.C** chiama una procedura definita nel modulo **ASMMOD1.ASM** per visualizzare una stringa **C** definita nello stesso modulo **CMOD1.C**; il modulo **ASMMOD1.ASM** chiama una procedura definita nel modulo **CMOD1.C** per visualizzare una stringa **C** definita nello stesso modulo **ASMMOD1.ASM**. In entrambi i casi la stringa **C** viene visualizzata attraverso la funzione **printf** della libreria standard del **C**; in questo modo possiamo anche vedere come sia possibile chiamare da un modulo **Assembly** le funzioni della libreria standard del **C**.
Il modulo **CMOD1.C** svolge il ruolo di modulo principale del programma, e quindi contiene anche l'**entry point** che, come vedremo in seguito, viene gestito direttamente dal compilatore **C**; il modulo **CMOD1.C** assume il seguente aspetto:
/* file cmod1.c */

```
#include < stdio.h >

/* variabili globali */

extern char strasm[];
char        strc[] = "Stringa definita nel modulo CMOD1.C";

/* prototipi di funzione */

extern void printstr_asm(void);
void printstr_c(void);

/* entry point */

int main(void)
{
   printstr_asm();

   return 0;
}

/* printstr_c */

void printstr_c(void)
{
   printf("%s\n", strasm);
}
```

Come si puo' notare, in questo modulo viene definita una stringa **strc** con linkaggio esterno, e viene dichiarata una stringa esterna **strasm**; inoltre viene definita una procedura **printstr_c** con linkaggio esterno, e viene dichiarata una procedura esterna **printstr_asm**. Appena si entra nella funzione principale **main**, viene chiamata la procedura esterna **printstr_asm** definita nel modulo **ASMMOD1.ASM**.

Analizziamo ora proprio il modulo **ASMMOD1.ASM** che essendo un modulo secondario, non deve contenere nessun **entry point**; la prima versione di questo modulo e' scritta in stile **Assembly** classico e non fa uso quindi delle caratteristiche avanzate. Come e' stato detto in precedenza, in un caso del genere siamo obbligati a specificare tutti i dettagli relativi ai segmenti standard; tutte le procedure devono indicare il loro tipo (**NEAR** o **FAR**) che deve rispecchiare il modello di memoria che vogliamo utilizzare. Naturalmente, la gestione dello **stack frame** e' interamente a carico del programmatore; anche la gestione delle direttive **ASSUME** e del gruppo **DGROUP** ricade sul programmatore. Nel caso di un programma con modello di memoria **SMALL**, il modulo **ASMMOD1.ASM** assume il seguente aspetto:

```
; file asmmod1.asm

; direttive per l'assembler

.386                            ; set di istruzioni a 32 bit

; segmento dati inizializzati NEAR

_DATA       SEGMENT  WORD PUBLIC USE16 'DATA'

   EXTRN    _strc: BYTE
   PUBLIC   _strasm

_strasm     db    'Stringa definita nel modulo ASMMOD1.ASM', 0
strFmt      db    '%s', 10, 0

_DATA       ENDS

; gruppo DGROUP

DGROUP      GROUP _DATA

; segmento principale di codice

_TEXT       SEGMENT  WORD PUBLIC USE16 'CODE'

   EXTRN    _printf: NEAR        ; funzione della libreria C
   EXTRN    _printstr_c: NEAR
   PUBLIC   _printstr_asm

   ASSUME   cs: _TEXT, ds: DGROUP

; void printstr_asm(void);

_printstr_asm proc near

   push     offset DGROUP:_strc  ; stringa da visualizzare
   push     offset DGROUP:strFmt ; stringa di formato
   call     near ptr _printf     ; chiama printf
   add      sp, 4                ; pulizia stack

   call     near ptr _printstr_c ; chiama il modulo C

   ret                           ; near return

_printstr_asm endp

_TEXT       ENDS

   END
```

La prima cosa da osservare riguarda il fatto che il **C** e' **case sensitive**, per cui tutti gli identificatori presenti nel modulo **ASMMOD1.ASM** devono letteralmente coincidere con i corrispondenti identificatori presenti nel modulo **CMOD1.C**; notiamo poi la presenza del carattere underscore all'inizio di ogni identificatore con linkaggio esterno.
In virtu' del fatto che stiamo utilizzando il modello di memoria **SMALL**, tutte le procedure interne o esterne sono di tipo **NEAR**; analogamente, l'accesso a tutte le variabili del programma avviene tramite indirizzi di tipo **NEAR**, costituiti quindi dalla sola componente **offset** a **16** bit.
La procedura **printstr_asm** chiamata dalla funzione **main** del modulo **CMOD1.C**, utilizza la funzione **printf** della libreria standard del **C** per visualizzare la stringa **strc** definita nel modulo **CMOD1.C**; osserviamo che la direttiva **EXTRN** per **printf** si trova nel blocco **_TEXT** in quanto nel modello **SMALL** esiste un solo blocco di codice. In sostanza, al momento di generare l'eseguibile, il linker osserva che abbiamo richiesto il modello di memoria **SMALL**, e quindi linka il nostro programma ad una apposita libreria **C** in formato **SMALL** contenete un blocco **_TEXT** all'interno del quale si trova la definizione della funzione **printf**; in generale comunque questo modo di ragionare potrebbe rivelarsi piuttosto azzardato in quanto i diversi compilatori **C** potrebbero anche utilizzare dei segmenti a noi sconosciuti per contenere le definizioni delle varie funzioni di libreria. Per questo motivo, ogni volta che dobbiamo dichiarare un identificatore definito in un segmento che non conosciamo, dobbiamo inserire la relativa direttiva **EXTRN** al di fuori di qualsiasi segmento di programma; in questo modo come sappiamo, deleghiamo al linker il compito di individuare la coppia **seg:offset** relativa all'identificatore stesso.
Il prototipo **C** della funzione **printf** e':

```
int printf(char *format, ...);
```

Questa funzione richiede quindi un numero variabile di argomenti, di cui il primo e' una stringa **C** chiamata **stringa di formato**; lo scopo della stringa **format** e' quello di formattare l'output prodotto da **printf**. Nel blocco **_DATA** del modulo **ASMMOD1.ASM** creiamo a tale proposito la stringa di formato **strFmt** (terminata da uno **0**), che chiede a **printf** di visualizzare una stringa (**%s**) seguita da un **new line** (**10='\n'**); la variabile **strFmt** ha linkaggio interno per cui non richiede l'underscore.
All'interno di **printstr_asm**, la chiamata di **printf** avviene in stile **Assembly** classico; di conseguenza, dobbiamo inserire gli argomenti nello stack a patire dall'ultimo, e alla fine dobbiamo provvedere a ripulire lo stack. Siccome avevamo inserito due offset da **16** bit ciascuno, la pulizia dello stack consiste nel sommare **4** byte a **SP**; si tenga anche presente che **printf** si aspetta che **strc** sia una stringa **C** con zero finale. Si noti la presenza del segment override **DGROUP:** che si rende necessario a causa del bug di **MASM** relativo all'uso dell'operatore **OFFSET** in presenza del gruppo **DGROUP**; come e' stato detto nel precedente capitolo, il **TASM** lavora in modalita' predefinita di emulazione del **MASM**, e questa emulazione e' cosi' "perfetta" che i progettisti del **TASM** hanno deciso di includere anche i bug del **MASM**.
Successivamente **printstr_asm** chiama la procedura **printstr_c** definita nel modulo **CMOD1.C**; all'interno del modulo **CMOD1.C**, la procedura **printstr_c** utilizza **printf** per visualizzare la stringa **strasm** definita nel modulo **ASMMOD1.ASM**. Anche in questo caso **strasm** deve essere una stringa **C** terminata da uno **0**; siccome questa stringa viene definita in un modulo **Assembly**, il compito di inserire lo zero finale spetta questa volta al programmatore.

Dalle considerazioni appena svolte emerge chiaramente il fatto che nell'interfacciamento dell'**Assembly** con i linguaggi di alto livello, il programmatore e' chiamato a svolgere un lavoro veramente impegnativo; infatti, oltre a dover scrivere il codice **Assembly**, il programmatore deve anche gestire una numerosa serie di dettagli legati ai protocolli di comunicazione tra i vari moduli.

Avendo a disposizione le versioni piu' recenti di **TASM** o di **MASM**, e' vivamente consigliabile servirsi delle caratteristiche avanzate fornite da questi potenti assembler; utilizzando le caratteristiche avanzate di **MASM** e **TASM**, il modulo **ASMMOD1.ASM** assume il seguente aspetto (in versione **TASM**):

```
; file asmmod1.asm

; direttive per l'assembler

.MODEL      SMALL, C            ; memory model & language
.386                            ; set di istruzioni a 32 bit

; prototipi delle procedure esterne

printf      PROTO  :WORD :?
printstr_c  PROTO

; variabili esterne

   EXTRN    strc: BYTE

; segmento dati inizializzati NEAR

.DATA

   PUBLIC   strasm

strasm      db    'Stringa definita nel modulo ASMMOD1.ASM', 0
strFmt      db    '%s', 10, 0

; segmento principale di codice

.CODE

   PUBLIC   printstr_asm

; void printstr_asm(void);

printstr_asm proc

   call     printf, offset strFmt, offset strc
   call     printstr_c

   ret                          ; near return

printstr_asm endp

   END
```

In questa nuova versione del modulo **ASMMOD1.ASM**, notiamo subito la scomparsa dei dettagli relativi alle caratteristiche dei segmenti di programma, del gruppo **DGROUP**, delle direttive **ASSUME** e degli underscores; in presenza della direttiva **.MODEL** che specifica il modello di memoria e il linguaggio di riferimento, tutti questi aspetti vengono gestiti direttamente dall'assembler. La presenza della direttiva **.MODEL** permette di delegare all'assembler anche i dettagli relativi al tipo **NEAR** o **FAR** delle procedure; in relazione agli identificatori pubblici, le necessarie direttive **PUBLIC** possono essere inserite all'inizio del programma, oppure all'interno dei segmenti contenenti le definizioni degli identificatori stessi.

In relazione invece agli identificatori esterni, come e' stato detto in precedenza si raccomanda vivamente di inserire le direttive **EXTRN** all'inizio del programma, al di fuori di qualsiasi segmento; in questo modo lasciamo al linker il compito di individuare la coppia **seg:offset** relativa ad ogni identificatore. Se si utilizza la direttiva **EXTRN** per le procedure esterne, bisogna indicare il tipo **PROC** al posto di **NEAR** o **FAR**; si puo' scrivere ad esempio:

```
EXTRN printf: PROC, printstr_c: PROC
```

In alternativa alla direttiva **EXTRN** si puo' anche usare la direttiva **PROTO**; il prototipo di **printf** in versione **TASM** e':

```
printf PROTO :WORD :?
```

Il prototipo di **printf** in versione **MASM** e':

```
printf PROTO :WORD, :VARARG
```

L'esempio appena presentato, pur essendo molto semplice, ci permette di analizzare alcuni aspetti molto delicati; come e' stato detto in precedenza, e' fondamentale che ci sia una perfetta corrispondenza tra definizioni e dichiarazioni relative agli identificatori con linkaggio esterno presenti nei vari moduli **C** e **Assembly**. Possiamo notare ad esempio che nel modulo **CMOD1.C**, la variabile **strc** e' stata definita come vettore di **BYTE**; di conseguenza, la direttiva **EXTRN** presente nel modulo **ASMMOD1.ASM**, deve dichiarare **strc** come variabile di tipo **BYTE**, cioe' come variabile che consente di accedere a delle locazioni di memoria da **8** bit. Se nel modulo **CMOD1.C** proviamo a modificare la definizione di **strc** scrivendo:

```
char *strc = "Stringa definita nel modulo CMOD1.C";
```

possiamo subito constatare che il programma non funziona piu' (o funziona in modo anomalo); questo accade in quanto nel modulo **ASMMOD1.ASM** la variabile esterna **strc** viene dichiarata di tipo **BYTE**, mentre nel modulo **CMOD1.C** la variabile **strc** viene definita come puntatore a **char**, e cioe' come indirizzo di una locazione di memoria da **8** bit. Nel modello di memoria **SMALL** un indirizzo e' formato dalla sola componente **offset** a **16** bit; di conseguenza, nel modulo **ASMMOD1.ASM** dobbiamo ridichiarare **strc** come:

```
EXTRN _strc: WORD
```

Non solo, ma tenendo anche conto del fatto che questa volta **strc** contiene gia' l'offset iniziale di un vettore di **char**, all'interno della procedura **printstr_asm** dobbiamo sostituire l'istruzione:

```
push offset DGROUP:_strc
```

con l'istruzione:

```
push _strc
```

Se stiamo utilizzando le caratteristiche avanzate di **MASM** e **TASM**, la chiamata:

```
call printf, offset strFmt, offset strc
```

diventa:

```
call printf, offset strFmt, strc
```

Lo stesso discorso vale naturalmente per la variabile **strasm**; se vogliamo che questa variabile sia un puntatore ad un vettore di **char**, nel modulo **ASMMOD1.ASM** possiamo scrivere:

```
   PUBLIC   _strasm

stringa_asm db    'Stringa definita nel modulo ASMMOD1.ASM', 0
_strasm     dw    offset stringa_asm
```

Come si puo' notare, la variabile **stringa_asm** ha un linkaggio di tipo interno per cui non necessita dell'underscore; a questo punto, nel modulo **CMOD1.C** dobbiamo ridichiarare **strasm** come:

```
extern char *strasm;
```

La funzione **printstr_c** non necessita di nessuna modifica in quanto in **C** come sappiamo, il nome di un vettore viene trattato come indirizzo iniziale del vettore stesso; di conseguenza la funzione **printf** riceve in ogni caso l'indirizzo iniziale della stringa **strasm**.

Un'altro aspetto interessante riguarda l'eventualita' di dover scavalcare il modello di memoria **SMALL** per poter utilizzare procedure di tipo **FAR**; anche in questo caso bisogna ricordarsi che le dichiarazioni e le definizioni nei vari moduli devono coincidere. Supponiamo ad esempio di voler ridefinire **printstr_asm** come procedura di tipo **FAR**; a tale proposito, nel modulo **ASMMOD1.ASM** l'intestazione di **printstr_asm** diventa:

```
_printstr_asm proc far
```

Di conseguenza, nel modulo **CMOD1.C** dobbiamo ridichiarare **printstr_asm** come:

```
extern void far printstr_asm(void);
```

Se non prendiamo questa precauzione, il programma va sicuramente in crash, e in molti casi puo' anche andare in crash l'intero **SO**.

La struttura del precedente programma puo' variare a seconda del modello di memoria utilizzato; in particolare, il programmatore deve prestare particolare attenzione agli indirizzamenti relativi ai dati del programma.
Nei modelli di memoria che comportano la presenza di un unico segmento di dati, tutti gli indirizzamenti per i dati sono di tipo **NEAR**; in un caso del genere infatti, per tutta la fase di esecuzione del programma abbiamo **DS=DGROUP** con **DGROUP** che raggruppa gli eventuali segmenti di dati **_DATA**, **_BSS**, **_CONST** e **_STACK**.
Nei modelli di memoria che comportano la presenza di un unico segmento di codice, tutti gli indirizzamenti per le procedure sono di tipo **NEAR**; in un caso del genere infatti, per tutta la fase di esecuzione del programma abbiamo **CS=_TEXT**.
Nei modelli di memoria che comportano la presenza di due o piu' segmenti di codice, tutti gli indirizzamenti predefiniti per le procedure sono di tipo **FAR**; in un caso del genere infatti, il caller puo' trovarsi in un segmento di programma diverso da quello della procedura chiamata. In generale quindi la chiamata di una procedura rende necessaria anche la modifica di **CS**; questa situazione e' gia' stata illustrata in precedenza, e in genere viene gestita automaticamente dall'assembler grazie all'apposito parametro specificato nella direttiva **.MODEL**.
Sicuramente il caso piu' interessante e delicato riguarda quei modelli di memoria che prevedono la presenza di due o piu' segmenti di dati; come gia' sappiamo, il compilatore inizializza **DS** ponendo in ogni caso **DS=DGROUP**. Si puo' presentare allora il caso della chiamata di una procedura che deve accedere con **DS** ad un segmento dati; se questo segmento dati appartiene a **DGROUP** non ci sono problemi in quanto non c'e' bisogno di modificare **DS**. I problemi invece si presentano quando la procedura deve accedere con **DS** ad un blocco **FAR_DATA**; in questo caso e' fondamentale che la procedura restituisca il controllo al caller solo dopo aver ripristinato **DS**.
Supponiamo ad esempio di riscrivere il modulo **ASMMOD1.ASM** per il modello di memoria **LARGE**, e supponiamo anche di avere in questo modulo il seguente blocco dati:

```
.FARDATA

farDword    dd    100000000
```

Se la procedura **printstr_asm** deve accedere per nome a **farDword**, si presenta il problema della modifica di **DS** che referenzia **DGROUP**; per evitare questa modifica, all'interno di **printstr_asm** potremmo utilizzare **ES** scrivendo:

```
mov      ax, @fardata
mov      es, ax
add      es:farDword, 200000000
```

In questo modo possiamo restituire il controllo alla funzione **main** del modulo **CMOD1.C** con **DS** che continua a referenziare **DGROUP**; se pero' vogliamo accedere ad una grande quantita' di dati presenti in **.FARDATA** e non vogliamo utilizzare i segment override (che tra l'altro aumentano le dimensioni del codice macchina), possiamo anche utilizzare **DS** scrivendo:

```
mov      ax, @fardata
mov      ds, @ax
assume   ds: @fardata

add      farDword, 200000000

; accesso alle altre variabili di .FARDATA

mov      ax, @data
mov      ds, ax
assume   ds: @data
```

Anche in questo caso, il registro **DS** viene restituito inalterato al caller; se non prendiamo questa precauzione, il crash del programma e' assicurato.

In presenza dei modelli di memoria come **COMPACT**, **LARGE** e **HUGE** che prevedono la presenza di due o piu' segmenti di dati, bisogna ricordarsi che l'accesso ai dati stessi potrebbe richiedere in questo caso indirizzi di tipo **FAR**; vediamo un esempio pratico che consiste nell'utilizzare il modello **LARGE** per il programma presentato in precedenza. Il modulo **CMOD1.C** rimane inalterato in quanto tutti i dettagli relativi agli indirizzamenti vengono gestiti dal compilatore **C**; in particolare, la funzione **printstr_c** diventa automaticamente **FAR**, e inoltre la funzione **printf** presente all'interno di **printstr_c** riceve questa volta come argomenti le coppie **seg:offset** relative sia alla stringa di formato, sia alla stringa **strasm** da visualizzare.

All'interno del modulo **ASMMOD1.ASM** invece, tutti questi aspetti ricadono sul programmatore; vediamo come deve essere modificato il modulo **ASMMOD1.ASM** che comprende anche il segmento **.FARDATA** descritto in precedenza:

```
; file asmmod1.asm

; direttive per l'assembler

.MODEL      LARGE, C              ; memory model & language
.386                              ; set di istruzioni a 32 bit

; prototipi delle procedure esterne

printf      PROTO  :WORD :?
printstr_c  PROTO

; variabili esterne

   EXTRN    strc: BYTE
```

```
; segmento dati inizializzati NEAR

.DATA

   PUBLIC   strasm

strasm      db    'Stringa definita nel modulo ASMMOD1.ASM', 0
strFmt      db    '%s', 10, 0
strFmt2     db    'farDword = %lu', 10, 0

; segmento dati inizializzati FAR

.FARDATA

farDword    dd    100000000

; segmento principale di codice

.CODE

   PUBLIC   printstr_asm

; void far printstr_asm(void);

printstr_asm proc

   call     printf, offset strFmt, ds, offset strc, ds
   call     printstr_c

   mov      ax, @fardata          ; accesso a .FARDATA
   mov      ds, ax                ; attraverso DS
   assume   ds: @fardata

   add      farDword, 20          ; farDword = farDword + 200000000
   mov      ebx, farDword         ; risultato in EBX

   mov      ax, @data             ; ripristino DS
   mov      ds, ax
   assume   ds: @data

   call     printf, offset strFmt2, ds, ebx

   ret                            ; far return

printstr_asm endp

   END
```

La presenza del parametro **LARGE** nella direttiva **.MODEL**, rende automaticamente **FAR** tutte le procedure prive di modificatore del modello di memoria; vediamo ora quello che succede all'interno di **printstr_asm**. La prima chiamata di **printf** serve per visualizzare la stringa **strc**; questa volta pero' **printf** per gli argomenti di tipo indirizzo richiede coppie **seg:offset**. Siccome gli argomenti **strFmt** e **strc** sono definiti sicuramente nel blocco **_DATA** referenziato da **DS**, possiamo scrivere:

```
call printf, offset strFmt, ds, offset strc, ds
```

E' importantissimo ricordare che in base alla convenzione **Intel**, gli indirizzi **FAR** formati da **16+16** bit devono contenere la componente **seg** nella **WORD** piu' significativa; siccome il **C** inserisce i parametri nello stack da destra verso sinistra, nella chiamata di **printf** dobbiamo posizionare la componente **seg** alla destra della corrispondente componente **offset**.

In questo modo la precedente direttiva **CALL** (o **INVOKE**) viene espansa in:

```
push    ds
push    offset strc
push    ds
push    offset strFmt
call    far ptr _printf
add     sp, 8
```

Come si puo' notare, per ogni coppia **seg:offset** viene inserita nello stack prima la componente **seg** e poi la componente **offset**; all'interno dello stack quindi, la componente **offset** precede la relativa componente **seg**.
La seconda chiamata di **printf** serve per visualizzare una variabile **farDword** di tipo **unsigned long**, definita nel blocco dati inizializzati **.FARDATA**; prima di tutto accediamo a questa variabile per nome, e a tale proposito utilizziamo **DS** per referenziare **.FARDATA**. Dopo aver modificato **farDword**, memorizziamo questa variabile in **EBX** e ripristiniamo **DS** in modo da fargli referenziare di nuovo **DGROUP**; a questo punto chiamiamo **printf** per visualizzare il contenuto di **EBX**. Per indicare a **printf** che **EBX** contiene un intero senza segno a **32** bit, dobbiamo utilizzare la stringa di formato **strFmt2**; questa stringa viene definita in **_DATA**, per cui e' fondamentale che **printf** riceva attraverso **DS** la giusta componente **seg** di **strFmt2**. In sostanza, se dopo aver modificato **farDword** dimentichiamo di ripristinare **DS**, la successiva chiamata di **printf** potrebbe anche mandare in crash il programma; si tenga presente inoltre che al termine di **printstr_asm** il controllo tornerebbe alla funzione **main** con **DS** che referenzia **.FARDATA**.

Generazione dell'eseguibile

Analizziamo ora le fasi di assembling e di linking dei precedenti programmi di esempio; queste fasi sono estremamente semplici grazie al fatto che i compilatori **C** producono per ogni modulo **C** il corrispondente **object file**; da queste considerazioni si intuisce subito che per convertire in formato eseguibile il programma precedente, sono necessari i seguenti semplici passi:

***** Compilazione di **CMOD1.C** per ottenere **CMOD1.OBJ**
***** Assemblaggio di **ASMMOD1.ASM** per ottenere **ASMMOD1.OBJ**
***** Linkaggio di **CMOD1.OBJ** e **ASMMOD1.OBJ** per ottenere **CMOD1.EXE**

Naturalmente, la fase di linking coinvolge anche le librerie contenenti le funzioni del **C**; i dettagli relativi alle varie fasi da svolgere, possono variare da compilatore a compilatore. Molto spesso i compilatori sono dotati di sofisticati ambienti integrati di sviluppo (**IDE**), che permettono di automatizzare tutto questo lavoro; nel caso ad esempio del celebre compilatore **Borland C++ 3.1** (che tra l'altro contiene anche la versione **3.x** del **TASM**), dall'interno dell'**IDE** e' possibile configurare tutte le opzioni come ad esempio il modello di memoria, il set di istruzioni della CPU, l'assembler da utilizzare, etc; una volta effettuate le varie configurazioni si puo' aprire un nuovo progetto che nel nostro caso sara' formato dai due moduli **CMOD1.C** e **ASMMOD1.ASM**.
Se non si dispone di un **IDE**, si puo' sempre ricorrere alla linea di comando che spesso rappresenta la soluzione piu' semplice ed efficace; in questo caso e' necessario inserire manualmente tutte le opzioni di configurazione richieste dal compilatore, dall'assemblatore e dal linker. In genere, digitando dal prompt del **DOS** il nome del compilatore e premendo **[Invio]**, si ottiene l'elenco delle opzioni del compilatore stesso; lo stesso discorso vale per l'assembler e per il linker.
Vediamo un caso pratico che si riferisce al compilatore **Borland C++ 3.1** a **16** bit; supponiamo a tale proposito che tutti gli strumenti di sviluppo siano stati installati nella cartella:
`c:\bc\bin`

In questo caso, tutte le librerie del **C** si trovano nella cartella:
`c:\bc\lib`
mentre gli include files si trovano nella cartella:
`c:\bc\include`
Prima di tutto bisogna creare nella cartella **BIN** i files di configurazione per il compilatore e per il linker; questi files devono essere in formato **ASCII** e si devono chiamare **TURBOC.CFG** e **TLINK.CFG**. All'interno del file **TURBOC.CFG** dobbiamo inserire i percorsi per le librerie, per gli include file e per l'assembler che vogliamo utilizzare, piu' eventuali parametri da passare al compilatore, al linker e all'assembler; l'aspetto generale del file **TURBOC.CFG** e' il seguente:

```
-Ic:\bc\include
-Lc:\bc\lib
-Ec:\tasm\bin\tasm32.exe
-Tml
```

Come si puo' notare, l'opzione **-E** permette di specificare l'assembler da utilizzare, mentre ogni opzione **-T** permette di passare un parametro di configurazione all'assembler stesso; questa tecnica non permette di specificare con l'opzione **-E** un assembler diverso da **TASM**. Se si intende utilizzare un compilatore **C** della **Borland** in combinazione con il **MASM**, bisogna separare la fase di compilazione dalla fase di assemblaggio; piu' avanti viene illustrato in pratica il procedimento da seguire.
L'aspetto generale del file **TLINK.CFG** e' il seguente:
`-Lc:\bc\lib`
Si tenga presente che spesso questi due files vengono creati automaticamente durante la fase di installazione; in questo caso il programmatore deve solo aggiungere le eventuali altre opzioni di cui ha bisogno. Ulteriori opzioni possono essere passate anche attraverso la linea di comando; ad esempio, l'opzione **-3** abilita il set di istruzioni a **32** bit del compilatore, mentre l'opzione **-f287** abilita l'uso della **FPU 80287/80387**.
A questo punto siamo pronti per la fase di creazione dell'eseguibile **CMOD1.EXE**; questa fase viene interamente gestita dal compilatore a linea di comando chiamato **BCC.EXE**. Per svolgere il suo lavoro il compilatore si serve proprio dei files di configurazione descritti in precedenza; e' importantissimo passare a **BCC.EXE** il parametro relativo al modello di memoria che intendiamo utilizzare. Questo parametro e' formato dall'opzione **-m** seguita dalla lettera iniziale del modello di memoria desiderato; se abbiamo deciso ad esempio di utilizzare il modello **SMALL** dobbiamo eseguire il comando:
`c:\bc\bin\bcc -ms -3 cmod1.c asmmod1.asm`
Se invece vogliamo utilizzare il modello **LARGE** dobbiamo eseguire il comando:
`c:\bc\bin\bcc -ml -3 cmod1.c asmmod1.asm`
Naturalmente il modello di memoria richiesto deve coincidere con quello specificato dalla direttiva **.MODEL** nel modulo **ASMMOD1.ASM**; a questo punto premendo il tasto **[Invio]** il controllo passa a **BCC.EXE** che:

***** compila **CMOD1.C** producendo **CMOD1.OBJ**;
***** assembla **ASMMOD1.ASM** producendo **ASMMOD1.OBJ**;
***** linka **CMOD1.OBJ + ASMMOD1.OBJ +** le librerie del **C**, producendo infine **CMOD1.EXE**.

Se vogliamo generare anche un dettagliatissimo **map file**, possiamo passare a **BCC.EXE** il parametro **-M**; in questo caso viene creato anche un file chiamato **CMOD1.MAP**.
Se vogliamo generare il **listing file** di **ASMMOD1.ASM**, dobbiamo passare a **BCC.EXE** il parametro **-Tl**; in questo caso viene creato anche un file chiamato **ASMMOD1.LST**.
Un parametro potentissimo e' sicuramente **-S** che richiede a **BCC.EXE** la generazione del file **CMOD1.ASM** che e' la traduzione in **Assembly** di **CMOD1.C** effettuata dal compilatore **C**; e' estremamente istruttivo esaminare il contenuto di **CMOD1.ASM** in quanto si ha la possibilita' di

verificare tutto il lavoro compiuto a basso livello dal compilatore **C**. L'esame di questo file permette anche di conoscere tutti i dettagli sul tipo di segmentazione utilizzato dal compilatore; si tenga presente che in presenza del parametro **-S**, non viene generato nessun eseguibile **CMOD1.EXE**.

Se abbiamo a disposizione un compilatore **C** della **Borland** e il **MASM** della **Microsoft**, dobbiamo procedere in questo modo:

Generazione del file **ASMMOD1.OBJ** con il comando:
```
ml /c /Cp asmmod1.asm
```
Generazione dell'eseguibile **CMOD1.EXE** con il comando (modello **SMALL**):
```
bcc -ms -3 cmod1.c asmmod1.obj
```
In assenza del parametro **/coff** il **MASM** genera un **object file** in formato **OMF** (Object Module Format) compatibile con gli **object files** generati dagli strumenti di sviluppo della **Borland**.

Funzioni che restituiscono strutture per valore

Passiamo ora ad un secondo esempio che illustra il caso di una funzione **C** che deve restituire una intera struttura per valore; se una funzione deve restituire per valore una struttura che occupa **3** byte oppure piu' di **4** byte, viene adottato un procedimento che comporta le fasi seguenti:

1) Il caller crea un'area di memoria sufficiente a contenere la struttura da restituire.
2) Il caller chiama la funzione passandole come ultimo argomento l'indirizzo **seg:offset** dell'area di memoria creata nella fase **1**.
3) La funzione chiamata utilizza **seg:offset** per riempiere la struttura da restituire.
4) La funzione termina restituendo **seg:offset** in **DX:AX**.

In sostanza, la funzione chiamata deve riempire una struttura che si trova in memoria all'indirizzo **seg:offset**; indipendentemente dal modello di memoria utilizzato, questo indirizzo deve essere sempre di tipo **FAR**. Si puo' notare che la fase **4** appare superflua in quanto il caller conosce gia' l'indirizzo **seg:offset**; in ogni caso si tratta di una convenzione del linguaggio **C** che il programmatore e' tenuto a seguire.
Se il caller si trova in un modulo **C**, le fasi **1** e **2** vengono gestite direttamente dal compilatore; se invece il caller si trova in un modulo **Assembly**, tutto questo lavoro ricade come al solito sul programmatore.
Per illustrare in pratica questi concetti, vediamo un esempio formato dai due moduli **CMOD2.C** e **ASMMOD2.ASM**; anche in questo esempio sono presenti chiamate incrociate tra i due moduli. Il modulo **CMOD2.C** definisce due strutture dati e chiama il modulo **ASMMOD2.ASM** per sommarle tra loro; il modulo **ASMMOD2.ASM** definisce due strutture dati e chiama il modulo **CMOD2.C** per sommarle tra loro.

Analizziamo prima di tutto la struttura del modulo **CMOD2.C**, che svolge come al solito il ruolo di modulo principale del programma:

```
/* file cmod2.c */

#include < stdio.h >

/* tipi e costanti */

typedef struct tagPoint3d {
   int      x;
   int      y;
   int      z;
} Point3d;

/* variabili globali */

static Point3d point1 = { 1000, 1100, 1200 };
static Point3d point2 = { 2000, 2100, 2200 };

/* prototipi di funzione */

extern Point3d sommapoint3d_asm(Point3d, Point3d);
Point3d sommapoint3d_c(Point3d, Point3d);

/* entry point */

int main(void)
{
   Point3d  psomma = sommapoint3d_asm(point1, point2);

   printf("MODULO C:   p.x = %d, p.y = %d, p.z = %d\n",
          psomma.x, psomma.y, psomma.z);

   return 0;
}

/* sommapoint3d_c */

Point3d sommapoint3d_c(Point3d p1, Point3d p2)
{
   p1.x += p2.x;
   p1.y += p2.y;
   p1.z += p2.z;

   return p1;
}
```

Questo modulo dichiara un nuovo tipo di dato **Point3d**, che rappresenta una struttura formata da tre membri di tipo **int** chiamati **x**, **y** e **z**; in totale, ciascun dato di tipo **Point3d** richiede quindi **6** byte di memoria.
In seguito vengono definite e inizializzate due variabili globali di tipo **Point3d**, chiamate **point1** e **point2**; come si puo' notare, queste due variabili hanno classe di memoria **static** per cui risultano invisibili all'esterno del modulo **CMOD2.C**.
Appena si entra nella funzione principale **main**, viene creata una nuova variabile di tipo **Point3d** chiamata **psomma**; naturalmente si tratta di una variabile locale che verra' quindi creata nello stack.

A questo punto **main** chiama la funzione esterna **sommapoint3d_asm** che ha il compito di sommare le due strutture **point1** e **point2**, restituendo poi il risultato alla funzione **main** che provvedera' a salvarlo nella struttura **psomma**; in questo esempio, per somma di due strutture **Point3d** si intende la somma dei membri corrispondenti (**x** con **x**, **y** con **y** e **z** con **z**).
In base alle cose dette in precedenza, quando il compilatore **C** incontra la chiamata di **sommapoint3d_asm**, la espande nelle seguenti istruzioni (modello di memoria **SMALL**):

```
push     point2.z
push     point2.y
push     point2.x
push     point1.z
push     point1.y
push     point1.x
push     seg psomma
lea      ax, psomma
push     ax
call     near ptr _sommapoint3d_asm
add      sp, 16
```

Come si puo' notare, il **C** inserisce nello stack i membri delle due strutture seguendo il solito ordine da destra a sinistra; alla fine il **C** inserisce "di nascosto" nello stack anche la coppia **seg:offset** relativa alla struttura **psomma** che rappresenta il valore di ritorno di **sommapoint3d_asm** (osserviamo che **psomma** e' una variabile locale, per cui il suo offset all'interno dello stack viene calcolato con **LEA**). Quando **sommapoint3d_asm** termina, la struttura **psomma** e' stata riempita con la somma tra **point1** e **point2**, e questo puo' essere verificato attraverso la **printf** che visualizza proprio i tre membri di **psomma**; si noti che il caller effettua la pulizia dello stack sommando ben **16** byte a **SP**.

Passiamo ora all'analisi del modulo **ASMMOD2.ASM**; anche in questo caso puo' essere interessante vedere quale struttura assume questo modulo quando si utilizza lo stile **Assembly** classico. Se stiamo utilizzando il modello di memoria **SMALL**, il modulo **ASMMOD2.ASM** avra' il seguente aspetto (versione **TASM**):

```
; file asmmod2.asm

; direttive per l'assembler

.386                             ; set di istruzioni a 32 bit

; tipi e costanti

Point3d  STRUC

   x     dw    ?
   y     dw    ?
   z     dw    ?

Point3d  ENDS

; identificatori esterni

   EXTRN    _printf: NEAR
   EXTRN    _sommapoint3d_c: NEAR

; segmento dati inizializzati NEAR

_DATA       SEGMENT  WORD PUBLIC USE16 'DATA'
```

```
point3      Point3d  < 4000, 4100, 4200 >
point4      Point3d  < 5000, 5100, 5200 >

strFmt      db       'MODULO ASM: p.x = %d, p.y = %d, p.z = %d', 10, 0

_DATA       ENDS

; segmento dati non inizializzati NEAR

_BSS        SEGMENT  WORD PUBLIC USE16 'BSS'

psomma      Point3d  < ?, ?, ? >

_BSS        ENDS

; gruppo DGROUP

DGROUP      GROUP    _DATA, _BSS

; segmento principale di codice

_TEXT       SEGMENT  WORD PUBLIC USE16 'CODE'

   PUBLIC   _sommapoint3d_asm

   ASSUME   cs: _TEXT, ds: DGROUP

; Point3d _sommapoint3d_asm(Point3d p1, Point3d p2);

_sommapoint3d_asm proc near

   p_addr   equ   [bp+4]         ; indirizzo struttura di ritorno
   p1       equ   [bp+8]         ; struct. p1
   p2       equ   [bp+14]        ; struct. p2

   push     bp                   ; preserva bp
   mov      bp, sp               ; ss:bp = ss:sp
   push     es                   ; preserva es

; psomma = sommapoint3d_c(point3, point4)

   push     point4.z
   push     point4.y
   push     point4.x
   push     point3.z
   push     point3.y
   push     point3.x
   push     ds
   push     offset DGROUP:psomma
   call     near ptr _sommapoint3d_c
   add      sp, 16

; visualizza psomma

   push     psomma.z
   push     psomma.y
   push     psomma.x
   push     offset DGROUP:strFmt
   call     near ptr _printf
   add      sp, 8

; return (p1 + p2)
```

```
   les      bx, p_addr          ; es:bx = p_addr

   mov      ax, p1[0]
   add      ax, p2[0]
   mov      es:[bx][0], ax      ; [p_addr.x] = p1.x + p2.x

   mov      ax, p1[2]
   add      ax, p2[2]
   mov      es:[bx][2], ax      ; [p_addr.y] = p1.y + p2.y

   mov      ax, p1[4]
   add      ax, p2[4]
   mov      es:[bx][4], ax      ; [p_addr.z] = p1.z + p2.z

   mov      dx, es
   mov      ax, bx              ; dx:ax = es:bx = p_addr

   pop      es                  ; ripristina es
   pop      bp                  ; ripristina bp
   ret                          ; near return

_sommapoint3d_asm endp

_TEXT       ENDS

   END
```

Anche **ASMMOD2.ASM** dichiara un nuovo tipo di dato **Point3d** che ha le stesse caratteristiche di quello dichiarato nel modulo **CMOD2.C**; e' importante ricordare che in qualsiasi linguaggio di programmazione, le dichiarazioni non occupano memoria in quanto sono solo dei modelli di dati o di procedure.
In seguito **ASMMOD2.ASM** definisce e inizializza nel blocco **_DATA** due variabili **point3** e **point4** di tipo **Point3d**; in assenza della direttiva **PUBLIC**, queste due variabili risultano invisibili all'esterno del modulo, e non necessitano quindi dell'underscore.
All'interno del blocco **_BSS** viene invece definita la variabile non inizializzata **psomma** di tipo **Point3d**; anche in questo caso abbiamo a che fare con una variabile con linkaggio interno, che non ha niente a che vedere quindi con la variabile locale **psomma** definita nel modulo **CMOD2.C**.
Siccome questo modulo e' scritto in **Assembly** classico, il programmatore e' tenuto a raggruppare **_DATA** e **_BSS** con la direttiva **GROUP**; notiamo inoltre che all'interno del blocco **_TEXT** e' necessaria una direttiva **ASSUME** che associa **DS** a **DGROUP**.
Per capire meglio quello che succede all'interno della procedura **sommapoint3d_asm**, e' necessario analizzare il significato che viene assunto in **Assembly** dai nomi dei membri di una struttura; consideriamo a tale proposito la variabile **point3** di tipo **Point3d**. Osserviamo subito che:

***** Il membro **x** si trova a **0** byte di distanza dall'offset iniziale di **point3**.
***** Il membro **y** si trova a **2** byte di distanza dall'offset iniziale di **point3**.
***** Il membro **z** si trova a **4** byte di distanza dall'offset iniziale di **point3**.

Quando la CPU incontra una istruzione del tipo:
`mov ax, point3.x`
trasferisce quindi in **AX** la **WORD** che si trova in memoria all'indirizzo che si ottiene sommando **0** byte all'offset iniziale di **point3**.

Quando la CPU incontra una istruzione del tipo:
```
mov ax, point3.y
```
trasferisce quindi in **AX** la **WORD** che si trova in memoria all'indirizzo che si ottiene sommando **2** byte all'offset iniziale di **point3**.
Quando la CPU incontra una istruzione del tipo:
```
mov ax, point3.z
```
trasferisce quindi in **AX** la **WORD** che si trova in memoria all'indirizzo che si ottiene sommando **4** byte all'offset iniziale di **point3**.
In sostanza, i membri **x**, **y** e **z** vengono trattati come se fossero variabili definite all'interno di un blocco dati chiamato **point3**; da queste considerazioni segue anche che l'istruzione:
```
mov ax, point3.x
```
equivale a:
```
mov ax, point3[x]
```
oppure a:
```
mov ax, [point3][x]
```
oppure a:
```
mov ax, [point3+x]
```
oppure a:
```
mov ax, [point3.x]
```
Naturalmente, al posto di **x**, **y** e **z** si possono utilizzare direttamente i corrispondenti spiazzamenti **0**, **2** e **4**; in questo caso e' proibito l'uso dell'operatore **.** (punto).
Se vogliamo accedere per indirizzo ad una variabile di tipo **Point3d**, siamo tenuti a gestire correttamente lo spiazzamento dei vari membri; se **BX** punta ad esempio all'indirizzo di **point3**, allora:
```
mov ax, [bx+0]
```
carica in **AX** il membro **x** di **point3**.
```
mov ax, [bx+2]
```
carica in **AX** il membro **y** di **point3**.
```
mov ax, [bx+4]
```
carica in **AX** il membro **z** di **point3**.
Come al solito, sono consentite anche le forme sintattiche esposte in precedenza; possiamo dire quindi che ad esempio:
```
mov ax, [bx+2]
```
puo' essere scritto anche come:
```
mov ax, [bx][2]
```
oppure:
```
mov ax, bx[2]
```
E' proibito invece scrivere:
```
mov ax, [bx.2]
```
Quando si utilizza il **TASM**, bisogna ricordare che i nomi dei membri di una struttura vengono considerati globali e quindi sono visibili anche all'esterno della struttura stessa; di conseguenza, nel modulo **ASMMOD2.ASM** non possiamo definire o dichiarare altri identificatori che utilizzano i nomi **x**, **y** e **z**.

Fatta questa premessa, ricordiamo che la funzione **main** del modulo **CMOD2.C** ha appena chiamato **sommapoint3d_asm** passandole come argomenti **point1** e **point2**; inoltre **main** ha passato un ulteriore argomento "nascosto" che rappresenta la coppia **seg:offset** della struttura **psomma** definita nello stesso modulo **CMOD2.C**. Siccome questo parametro e' stato passato per ultimo, si verra' a trovare nello **stack frame** di **sommapoint3d_asm** all'indirizzo **SS:BP+4** e occupera' **4** byte; di conseguenza, la copia **p1** di **point1** (da **6** byte) si viene a trovare a **SS:BP+8**, mentre la copia **p2** di **point2** (da **6** byte) si viene a trovare a **SS:BP+14**.
Il primo compito svolto da **sommapoint3d_asm** consiste nel chiamare la funzione **sommapoint3d_c** definita in **CMOD2.C**, per effettuare la somma tra **point3** e **point4** che verra' memorizzata nella struttura **psomma** definita nel modulo **ASMMOD2.ASM**; questa volta e'

compito del programmatore passare a **sommapoint3d_c** la coppia **seg:offset** della "struttura di ritorno" **psomma**. A tale proposito si puo' notare che vengono inseriti nello stack anche il segmento **DS=DGROUP** di **psomma**, e il suo offset calcolato rispetto a **DGROUP**; ricordiamo che il **TASM** simula il bug dell'operatore **OFFSET** del **MASM**, per cui in presenza del gruppo **DGROUP** e' necessario usare il segment override.
La funzione **sommapoint3d_c** somma direttamente **p2** a **p1** e copia poi il contenuto di **p1** nella struttura di ritorno **psomma** (definita in **ASMMOD2.ASM**); ricordiamo che nel passaggio degli argomenti per valore, vengono create nello stack delle copie degli argomenti stessi, che non hanno niente a che vedere con gli originali. Nel nostro caso, il passaggio per valore a **sommapoint3d_c** degli argomenti **point3** e **point4**, comporta la creazione nello stack delle copie **p1** e **p2** degli argomenti stessi; sommando **p2** a **p1**, non provochiamo quindi nessuna modifica all'argomento originale **point3**.
Per dimostrare che che **sommapoint3d_c** ha veramente sommato **point3** con **point4** restituendo il risultato in **psomma**, utilizziamo come al solito la **printf** del **C**; come si puo' notare, la **printf** utilizza la stringa di formato **strFmt** per visualizzare i tre membri di **psomma**.

Il compito successivo svolto da **sommapoint3d_asm** consiste nel sommare le copie di **point1** e **point2** passate per valore da **main**; prima di tutto carichiamo in **ES:BX** la coppia **seg:offset** della struttura di ritorno. L'istruzione utilizzata e':

```
les bx, p_addr
```

E' una buona abitudine preservare sempre il contenuto dei registri di segmento come **ES**, anche se non vengono utilizzati dal compilatore **C**.
A questo punto effettuiamo le varie somme con istruzioni del tipo:

```
mov      ax, p1[2]
add      ax, p2[2]
mov      es:[bx][2], ax
```

Osserviamo che in questo caso **p1** e **p2** sono della macro alfanumeriche e non delle variabili di tipo **Point3d**; e' necessario quindi analizzare attentamente il significato di istruzioni come:

```
mov ax, p1[2]
```

p1 e' una macro alfanumerica che rappresenta il corpo **[bp+8]**; di conseguenza, l'istruzione precedente viene espansa in:

```
mov ax, [bp+8][2]
```

che in sostanza equivale a:

```
mov ax, [bp+8][2] = mov ax, [bp+(8+2)] = mov ax, [bp+10]
```

In un caso di questo genere, non avrebbe nessun senso scrivere:

```
mov ax, [p1][2]
```

infatti, questa istruzione verrebbe espansa in:

```
mov ax, [[bp+8]][2]
```

che pur non avendo nessun significato, non produce nessun messaggio di errore da parte dell'assembler.
Un'ultima osservazione riguarda il fatto che **sommapoint3d_asm** prima di terminare carica in **DX:AX** l'indirizzo **p_addr** della struttura di ritorno.

Come ci dobbiamo comportare se vogliamo creare nello stack la struttura **psomma** definita nel blocco **_BSS** ?
Prima di tutto dobbiamo modificare lo **stack frame** di **sommapoint3d_asm** in modo da fare posto nello stack ai **6** byte di una struttura **Point3d**; a tale proposito dobbiamo scrivere:

```
push     bp       ; preserva bp
mov      bp, sp   ; ss:bp = ss:sp
sub      sp, 6    ; spazio per un Point3d
```

Per accedere alla variabile locale **psomma** creata nello stack possiamo scriverci la macro:
`psomma equ [bp-6]`
A questo punto la chiamata di **sommapoint3d_c** diventa:

```
push     point4.z
push     point4.y
push     point4.x
push     point3.z
push     point3.y
push     point3.x
push     ss
lea      ax, psomma
push     ax
call     near ptr _sommapoint3d_c
add      sp, 16
```

Questa volta **psomma** e' una macro e non una variabile di tipo **Point3d**; di conseguenza, la successiva chiamata a **printf** diventa:

```
push     psomma[4]
push     psomma[2]
push     psomma[0]
push     offset DGROUP:strFmt
call     near ptr _printf
add      sp, 8
```

Osserviamo che grazie al linkaggio interno, il nome **psomma** del modulo **CMOD2.C** non entra in conflitto con il nome **psomma** del modulo **ASMMOD2.ASM**; un'ultima considerazione importante riguarda il fatto che prima di uscire da **sommapoint3d_asm**, bisogna ricordarsi di ripristinare **SP** per togliere dallo stack le variabili locali.
Il programma appena illustrato dimostra ulteriormente che nell'interfacciamento tra l'**Assembly** e i linguaggi di alto livello, il programmatore e' chiamato a svolgere un vero e proprio superlavoro; se vogliamo scaricare sull'assembler una buona parte di questa fatica, possiamo ricorrere come al solito alle caratteristiche avanzate di **MASM** e **TASM**.

Nell'ipotesi di voler utilizzare il modello di memoria **LARGE**, il modulo **ASMMOD2.ASM** assume il seguente aspetto (versione **MASM**):

```
; file asmmod2.asm

; direttive per l'assembler

.MODEL            LARGE, C
.386

; tipi e costanti

Point3d  STRUC

   x     dw    ?
   y     dw    ?
   z     dw    ?

Point3d  ENDS

; prototipi delle procedure

printf            PROTO :WORD, :VARARG
sommapoint3d_c    PROTO :WORD, :WORD, :Point3d, :Point3d

; segmento dati inizializzati NEAR

.DATA

point3      Point3d  < 4000, 4100, 4200 >
point4      Point3d  < 5000, 5100, 5200 >

strFmt      db       'MODULO ASM: p.x = %d, p.y = %d, p.z = %d', 10, 0

; segmento dati non inizializzati NEAR

.DATA?

psomma      Point3d  < ?, ?, ? >

; segmento principale di codice

.CODE

   PUBLIC   sommapoint3d_asm

; Point3d _sommapoint3d_asm(Point3d p1, Point3d p2);

sommapoint3d_asm proc p_addr :DWORD, p1 :Point3d, p2 :Point3d

   push     es                   ; preserva es

; psomma = point3 + point4

   invoke   sommapoint3d_c, offset psomma, ds, point3, point4
   invoke   printf, offset strFmt, ds, psomma.x, psomma.y, psomma.z

; return (p1 + p2)

   les      bx, p_addr           ; es:bx = p_addr

   mov      ax, p1.x
```

```
   add      ax, p2.x
   mov      es:[bx][0], ax       ; [p_addr.x] = p1.x + p2.x

   mov      ax, p1.y
   add      ax, p2.y
   mov      es:[bx][2], ax       ; [p_addr.y] = p1.y + p2.y

   mov      ax, p1.z
   add      ax, p2.z
   mov      es:[bx][4], ax       ; [p_addr.z] = p1.z + p2.z

   mov      dx, es
   mov      ax, bx               ; dx:ax = es:bx = p_add

   pop      es                   ; ripristina es
   ret                           ; far return

sommapoint3d_asm endp

   END
```

In questa nuova versione del modulo **ASMMOD2.ASM**, notiamo una notevole riduzione della quantita' di codice che il programmatore deve inserire; naturalmente, una volta che l'assembler ha svolto il proprio lavoro, si riottiene la stessa situazione mostrata nella versione classica del modulo **ASMMOD2.ASM**.

Come e' stato specificato in precedenza, questa nuova versione di **ASMMOD2.ASM** utilizza la sintassi avanzata del **MASM**; generiamo quindi **ASMMOD2.OBJ** con il comando:
`ml /c /Cp asmmod2.asm` (**/c** = compile only, **/Cp** = case sensitive).
Generiamo poi **CMOD2.EXE** con il comando:
`bcc -ml -3 cmod2.c asmmod2.obj` (**-ml** = model large).

Interscambio di variabili di tipo Floating Point

Analizziamo ora un esempio che illustra l'interscambio di numeri in floating point tra moduli **C** e moduli **Assembly**; i numeri in floating point possono essere gestiti sia via hardware che via software. Per la gestione via hardware e' necessario un coprocessore matematico (**FPU**) che ormai si trova incorporato su tutte le CPU **80486** e superiori; tutti i dettagli sull'uso della **FPU** vengono esposti in un apposito capitolo della sezione **Assembly Avanzato**. Se non si dispone di una **FPU** e' necessario ricorrere ad apposite procedure contenenti algoritmi di simulazione che permettono alla normale **CPU** di gestire (molto piu' lentamente) i numeri reali; anche questi aspetti vengono illustrati nella sezione **Assembly Avanzato**. I compilatori **C** forniscono una libreria matematica standard che sfrutta l'eventuale presenza di una **FPU**; in assenza della **FPU** viene utilizzata una apposita libreria di emulazione fornita ugualmente dal compilatore.
Per il semplice esempio illustrato piu' avanti e' sufficiente sapere che la **FPU** dispone di **8** registri a **80** bit chiamati **ST(0)**, **ST(1)**, **ST(2)**, **ST(3)**, **ST(4)**, **ST(5)**, **ST(6)**, **ST(7)**; questi **8** registri formano una struttura a stack di cui **ST(0)** e' la cima (**TOS**). Ogni volta che si carica un numero (intero o reale) nella **FPU**, questo numero viene convertito in un formato chiamato **temporary real** a **80** bit e viene sistemato in **ST(0)**; il vecchio contenuto di **ST(0)** scala di un posto e si porta in **ST(1)**, il vecchio contenuto di **ST(1)** scala di un posto e si porta in **ST(2)**, e cosi' via.

Le istruzioni della **FPU** iniziano tutte con la lettera **F** che significa **fast** (veloce); nel nostro esempio utilizziamo le seguenti istruzioni:

```
FINIT:   inizializza la FPU e azzera gli 8 registri dello stack
FLD op:  carica l'operando op in ST(0)
FMUL op: moltiplica l'operando op per ST(0) e pone il risultato in ST(0)
FADD:    somma ST(0) + ST(1) e pone il risultato in ST(0)
FSQRT:   calcola la radice quadrata di ST(0) e pone il risultato in ST(0)
FSTP op: estrae il contenuto di ST(0) e lo copia nell'operando op
FWAIT:   sincronizza la FPU con la CPU
```

L'istruzione **FWAIT** era necessaria sui vecchi computers dove poteva capitare che la **CPU** tentasse di accedere ad una locazione di memoria nella quale la **FPU** doveva ancora salvare il risultato di un calcolo; con le CPU **80486** e superiori, questa istruzione non e' piu' necessaria e il suo utilizzo viene ignorato. Le istruzioni **FLD**, **FMUL** e **FSTP** presuppongono che l'operando **op** sia un numero reale in formato **IEEE**.

Il programma di esempio e' formato dai due moduli **CMOD3.C** e **ASMMOD3.ASM**; analizziamo prima di tutto il modulo **CMOD3.C** che ricopre il ruolo di modulo principale del programma:

```
/* file cmod3.c */

#include < stdio.h >
#include < math.h >

static double  lato1 = 354.004412657835126;
static double  lato2 = 128.156734523337698;

/* prototipi di funzione */

extern double diagonale_asm(double, double);
double diagonale_c(double, double);

/* entry point */

int main(void)
{
   double   diag = diagonale_asm(lato1, lato2);

   printf("MODULO C:   Diagonale = %.15f\n", diag);

   return 0;
}

/* diagonale_c */

double diagonale_c(double d1, double d2)
{
   return sqrt((d1 * d1) + (d2 * d2));
}
```

Come si puo' notare, il modulo **CMOD3.C** definisce due variabili **lato1** e **lato2** di tipo **double** (numero reale a **64** bit); queste due variabili sono **static** per cui risultano invisibili all'esterno. All'interno della funzione principale **main** viene chiamata la funzione esterna **diagonale_asm** che riceve **lato1** e **lato2** come argomenti; **diagonale_asm** utilizza il teorema di Pitagora per calcolare la diagonale del rettangolo che ha **lato1** e **lato2** come lati. Il risultato ottenuto viene restituito come valore di ritorno; in sostanza, viene effettuato il calcolo:

`diagonale = SQRT((lato1 * lato1) + (lato2 * lato2))` (**SQRT** = radice quadrata)
Per capire come lavora **diagonale_asm**, dobbiamo analizzare il modulo **ASMMOD3.ASM**:

```
; file asmmod3.asm

; direttive per l'assembler

.386                                ; set di istruzioni a 32 bit

; identificatori esterni

   EXTRN    _printf: NEAR
   EXTRN    _diagonale_c: NEAR

; segmento dati inizializzati NEAR

_DATA       SEGMENT  WORD PUBLIC USE16 'DATA'

lato3       dq       231.846375236468923
lato4       dq       625.236555681945232

strFmt      db       'MODULO ASM: Diagonale = %.15f', 10, 0

_DATA       ENDS

; gruppo DGROUP

DGROUP      GROUP    _DATA

; segmento principale di codice

_TEXT       SEGMENT  WORD PUBLIC USE16 'CODE'

   PUBLIC   _diagonale_asm

   ASSUME   cs: _TEXT, ds: DGROUP

; double _diagonale_asm(double d1, double d2);

_diagonale_asm proc near

   d1       equ   [bp+4]         ; double d1
   d2       equ   [bp+12]        ; double d2
   diag     equ   [bp-8]         ; double diag

   push     bp                   ; preserva bp
   mov      bp, sp               ; ss:bp = ss:sp
   sub      sp, 8                ; 8 byte di variabili locali

; diag = _diagonale_c(lato3, lato4)

   finit                         ; inizializza la FPU
   push     dword ptr lato4[4]
   push     dword ptr lato4[0]
   push     dword ptr lato3[4]
   push     dword ptr lato3[0]
   call     near ptr _diagonale_c
   add      sp, 16
   fstp     qword ptr diag       ; copia ST(0) in diag
   fwait                         ; sincronizzazione

; visualizza diag
```

```
   push     dword ptr diag[4]
   push     dword ptr diag[0]
   push     offset DGROUP:strFmt
   call     near ptr _printf
   add      sp, 10


; ST(0) = SQRT(d1^2 + d2^2)

   fld      qword ptr d1          ; ST(0) = d1
   fmul     qword ptr d1          ; ST(0) = d1 * d1
   fld      qword ptr d2          ; ST(0) = d2
   fmul     qword ptr d2          ; ST(0) = d2 * d2
   fadd                           ; ST(0) = ST(0) + ST(1)
   fsqrt                          ; ST(0) = SQRT(ST(0))
   fwait                          ; sincronizzazione

   mov      sp, bp                ; ripristina sp
   pop      bp                    ; ripristina bp
   ret                            ; near return

_diagonale_asm endp

_TEXT       ENDS

   END
```

Anche nel modulo **ASMMOD3.ASM** vengono definite due variabili **lato3** e **lato4** di tipo **QWORD**, cioe' **double** a **64** bit; in assenza della direttiva **PUBLIC** queste due variabili risultano invisibili all'esterno.
La procedura **diagonale_asm** chiamata da **main**, chiama a sua volta la funzione esterna **diagonale_c** definita nel modulo **CMOD3.C**; questa funzione riceve come argomenti **lato3** e **lato4**, e restituisce la diagonale del rettangolo che ha questi due argomenti come lati. A tale proposito si puo' notare che **diagonale_c** utilizza la funzione **sqrt** appartenente alla libreria matematica del **C**; notiamo infatti che all'inizio del modulo **CMOD3.C** e' presente l'inclusione dell'header standard **math.h**.
Come e' stato detto all'inizio del capitolo, i valori in floating point restituiti da una funzione, si trovano nel registro **ST(0)** della **FPU**; i registri della **FPU** sono a **80** bit, per cui possono gestire **float** a **32** bit, **double** a **64** bit e **long double** a **80** bit. Nel nostro caso, con l'istruzione **FSTP** estraiamo un **double** da **ST(0)** e lo salviamo nella variabile locale **diag** a **64** bit creata nello stack da **diagonale_asm**; osserviamo infatti che vengono sottratti **8** byte al registro **SP** per fare posto a **diag**.
Successivamente il contenuto di **diag** viene visualizzato con **printf**; si noti che la stringa di formato **strFmt** prevede la visualizzazione di un float con almeno **15** cifre dopo la virgola in modo da dare la possibilita' di analizzare la precisione del risultato. Per le verifiche si puo' utilizzare ad esempio la calcolatrice di **Windows** (in modalita' scientifica); si tenga presente che tanto maggiore e' la parte intera del numero, tanto minore sara' la precisione della parte decimale.
Il compito finale svolto da **diagonale_asm** consiste nel ripetere lo stesso identico calcolo in relazione ai due parametri **d1** e **d2** ricevuti da **main**; osserviamo che alla fine del calcolo, il risultato finale e' a disposizione di **main** nel registro **ST(0)** della **FPU**.

E' necessario fare molta attenzione al metodo seguito per l'inserimento nello stack delle **QWORD** da passare alle funzioni come **diagonale_c** e **printf**; grazie all'uso del set di istruzioni a **32** bit, possiamo utilizzare operandi a **32** bit con l'istruzione **PUSH**. Siccome stiamo lavorando con dati da **64** bit, dobbiamo inserire nello stack prima la **DWORD** piu' significativa e poi quella meno

significativa; se avessimo utilizzato il set di istruzioni a **16** bit, l'inserimento nello stack di un dato come **lato3** si sarebbe svolto nel modo seguente:

```
push     word ptr lato3[6]
push     word ptr lato3[4]
push     word ptr lato3[2]
push     word ptr lato3[0]
```

Come e' stato spiegato in un precedente capitolo, il **TASM** presenta una caratteristica veramente potente, che permette di trattare le istruzioni della CPU come se fossero delle macro; con il **TASM**, anche in presenza del set di istruzioni a **16** bit, e' possibile ugualmente srivere istruzioni del tipo:

```
push qword ptr lato3
```

Quando l'assembler incontra questa istruzione, la espande nelle quattro istruzioni viste in precedenza; questa caratteristica non e' disponibile con il **MASM**.
In ogni caso, se non vogliamo impazzire con tutti questi dettagli, possiamo sempre ricorrere alle carattersitiche avanzate di **MASM** e **TASM**; in questo caso, il modulo **ASMMOD3.ASM** assume il seguente aspetto (versione **MASM**):

```
; direttive per l'assembler

.MODEL   SMALL, C
.386

; prototipi delle procedure

printf      PROTO :WORD, :VARARG
diagonale_c PROTO :QWORD, :QWORD

; segmento dati inizializzati NEAR

.DATA

lato3       dq       231.846375236468923
lato4       dq       625.236555681945232

strFmt      db       'MODULO ASM: Diagonale = %.15f', 10, 0

; segmento principale di codice

.CODE

   PUBLIC   diagonale_asm

; double diagonale_asm(double d1, double d2);

diagonale_asm proc d1 :QWORD, d2 :QWORD

   LOCAL    diag  :QWORD

; diag = diagonale_c(lato3, lato4)

   finit                        ; inizializza la FPU
   invoke   diagonale_c, lato3, lato4
   fstp     qword ptr diag      ; salva ST(0) in diag
   fwait                        ; sincronizzazione

; visualizza diag

   invoke   printf, offset strFmt, diag
```

```
; ST(0) = SQRT(d1^2 + d2^2)

   fld      qword ptr d1        ; ST(0) = d1
   fmul     qword ptr d1        ; ST(0) = d1 * d1
   fld      qword ptr d2        ; ST(0) = d2
   fmul     qword ptr d2        ; ST(0) = d2 * d2
   fadd                         ; ST(0) = ST(0) + ST(1)
   fsqrt                        ; ST(0) = SQRT(ST(0))
   fwait                        ; sincronizzazione

   ret                          ; near return

diagonale_asm endp

   END
```

Anche in questo caso, supponendo di avere a disposizione il compilatore **BCC.EXE** della **Borland** e l'assemblatore **ML.EXE** della **Microsoft**, possiamo procedere in questo modo:

1) Generiamo **ASMMOD3.OBJ** con il comando:
`ml /c /Cp asmmod3.asm` (**/c** = compile only, **/Cp** = case sensitive)
2)Generiamo **CMOD3.EXE** con il comando:
`bcc -ms -3 -f287 cmod3.c asmmod3.obj` (**-ms** = model small, **-f287** = FPU 80287/80387).

Allineamento dei dati al BYTE e alla WORD

Molti compilatori **C** a **16** bit permettono al programmatore di selezionare l'allineamento dei dati al **BYTE** o alla **WORD**; chiaramente lo scopo di questo allineamento e' quello di ottimizzare l'accesso ai dati da parte delle CPU con **Data Bus** formato da **16** o piu' linee. Le modalita' per la richiesta del tipo di allineamento variano da compilatore a compilatore; nel caso ad esempio del **Borland C++ 3.1** e' necessario selezionare il menu **Options - Compiler - Code generation - Word alignment**. Alternativamente e' possibile passare l'opzione **-a** dalla linea di comando; per gli altri compilatori si possono consultare i relativi manuali (o l'help in linea).
Consideriamo ad esempio il seguente blocco dati di un programma **C**:

```
int      i1 = 10
char     c1 = 20;
int      i2 = 30;
```

Se e' attivo l'allineamento al **BYTE**, la variabile **i1** a **16** bit viene disposta all'offset **0000h** del blocco dati, la variabile **c1** a **8** bit viene disposta all'offset **0002h**, mentre la variabile **i2** a **16** bit viene disposta all'offset **0003h**; in questo caso **i2** parte da un offset dispari e la CPU ha bisogno di **2** accessi in memoria per leggere o scrivere nella relativa locazione di memoria a **16** bit.
Se invece e' attivo l'allineamento alla **WORD**, la variabile **i1** a **16** bit viene disposta all'offset **0000h** del blocco dati, la variabile **c1** a **8** bit viene disposta all'offset **0002h**, mentre la variabile **i2** a **16** bit viene disposta all'offset **0004h**; in questo caso **i2** parte da un offset pari, e la CPU ha bisogno di **1** solo accesso in memoria per leggere o scrivere nella relativa locazione di memoria da **16** bit.
Nei moduli **C** tutti questi aspetti vengono gestiti direttamente dal compilatore; nei moduli **Assembly** e' compito del programmatore tener conto dell'attributo di allineamento che e' stato selezionato per i dati **EXTRN** definiti in un modulo **C**. In generale, e' vivamente sconsigliabile scrivere programmi **Assembly** che cercano di dedurre in modo empirico l'offset di una variabile; utilizzando invece l'istruzione **LEA** o l'operatore **OFFSET** si ottiene questa informazione in modo assolutamente sicuro ed affidabile.

29.7 Argomenti dalla linea di comando

Un programma scritto in **ANSI C** deve contenere una funzione chiamata **main** che rappresenta il punto in cui il programmatore riceve il controllo; il prototipo standard della funzione **main** e':
`int main(int argc, char *argv[]);`
Il parametro **argc** e' un intero che rappresenta il numero di eventuali argomenti passati al programma dalla linea di comando; il parametro **argv** rappresenta un vettore di puntatori a stringhe, dove ogni stringa e' uno degli eventuali argomenti passati al programma dalla linea di comando. Per convenzione, **argv[0]** deve puntare al nome del programma completo di percorso; analogamente, **argv[argc]** deve puntare a **NULL**.
Possiamo verificare tutti questi aspetti attraverso il seguente esempio pratico:

```
/* file argvect.c */

#include < stdio.h >

int main(int argc, char *argv[])
{
   int   i;

   for (i = 0; i < argc; i++)
      printf("argv[%d] = %s\n", i, argv[i]);

   return 0;
}
```

Per verificare l'output prodotto da questo programma si puo' eseguire il comando:
`argvect arg1 arg2 arg3 arg4`

Il fatto che **main** riceva una lista di argomenti, ci fa capire chiaramente che questo non e' il vero **entry point** del programma; il vero **entry point** infatti si trova in un apposito modulo che in fase di generazione dell'eseguibile, verra' collegato dal linker al nostro programma. Nel caso ad esempio del **Borland C++ 3.1**, questo modulo viene chiamato **C0.ASM** e si trova nella cartella:
`c:\bc\lib\startup`
Il compito di questo modulo **Assembly** e' quello di effettuare una serie di inizializzazioni che comprendono ad esempio la creazione dello stack, la raccolta degli eventuali argomenti passati dalla linea di comando, etc; una volta terminate tutte queste fasi, il modulo **C0.ASM** chiama una funzione esterna che deve chiamarsi **_main**. A questa funzione vengono passati proprio gli argomenti **argc** e **argv** descritti in precedenza; per dimostrarlo, riscriviamo l'esempio **ARGVECT.C** in versione **Assembly**:

```
; file argvect.asm

.386                             ; set di istruzioni a 32 bit

; identificatori esterni

   EXTRN    __setargv__: FAR     ; inizializza argc, argv
   EXTRN    _printf : NEAR       ; funzione di libreria

; segmento dati inizializzati NEAR

_DATA       SEGMENT  WORD PUBLIC USE16 'DATA'

count       dw       0
strFmt      db       'argv[%d] = %s', 10, 0
```

```
_DATA       ENDS

; gruppo DGROUP

DGROUP      GROUP    _DATA

; segmento principale di codice

_TEXT       SEGMENT  WORD PUBLIC USE16 'CODE'

   PUBLIC   _main

   ASSUME   cs: _TEXT, ds: DGROUP

; int main(int argc, char *argv[])

_main proc near

   argc     equ   [bp+4]         ; parametro argc
   argv     equ   [bp+6]         ; parametro argv

   push     bp                   ; preserva bp
   mov      bp, sp               ; ss:bp = ss:sp
   push     si                   ; preserva si

   mov      si, argv             ; si = argv
arg_loop:
   push     word ptr [si]        ; argv[i]
   push     count                ; indice
   push     offset DGROUP:strFmt ; stringa di formato
   call     near ptr _printf     ; chiama _printf
   add      sp, 6                ; pulizia stack

   add      si, 2                ; prossimo argv
   inc      count                ; incremento indice
   dec      word ptr argc        ; decremento argc
   jnz      arg_loop             ; controllo loop

   pop      si                   ; ripristina si
   pop      bp                   ; ripristina bp
   ret                           ; near return

_main endp

_TEXT       ENDS

   END
```

Per ottenere l'eseguibile **ARGVECT.EXE** dobbiamo specificare il percorso del **TASM** nel file **TURBOC.CFG** e impartire il comando:

```
bcc -ms -3 argvect.asm
```

Come si puo' notare, e' necessario inserire una direttiva **EXTRN** relativa alla funzione **_setargv__** che si occupa dell'inizializzazione di **argc** e **argv**; trattandosi di un identificatore **FAR**, questa direttiva va inserita al di fuori di qualsiasi segmento di programma. Questi aspetti sono relativi esclusivamente al compilatore **Borland C++ 3.1**; altri compilatori possono anche richiedere un procedimento differente.
Con il modello di memoria **SMALL**, all'interno di **main** troviamo **argc** a **BP+4** e **argv** a **BP+6**; il loop **arg_loop** esegue lo stesso lavoro mostato nel precedente esempio **ARGVECT.C**. Osserviamo che per definizione, **argv** e' l'indirizzo iniziale di un vettore di indirizzi; in pratica, ogni elemento di **argv** e' l'indirizzo di una stringa. Caricando **argv** in **SI** possiamo dire che **[SI]** e' l'indirizzo della

prima stringa, **[SI+2]** e' l'indirizzo della seconda stringa, **[SI+4]** e' l'indirizzo della terza stringa, e cosi' via.
E' importante notare come la chiamata di **printf** non danneggia il contenuto di **SI**; ricordiamo infatti che tutte le funzioni del **C** preservano il contenuto di **SI** e di **DI** (in ogni caso sarebbe meglio non fidarsi).

Nell'ambiente operativo a **16** bit del **DOS**, gli argomenti che passiamo ad un programma attraverso la linea di comando, vengono inseriti dal **SO** nel **PSP** del programma stesso; se analizziamo la Figura 6 del Capitolo 13, possiamo notare infatti che esistono due membri della struttura **PSP** chiamati **CommandLineParmLength** e **CommandLineParameters**. Il primo membro e' un **BYTE** che si trova allo spiazzamento **80h**, e contiene la lunghezza della stringa comprendente tutti i parametri passati al programma; il secondo membro e' un vettore di **127 BYTE** che parte dallo spiazzamento **81h**, e contiene la stringa con tutti i parametri passati al programma. Per ricavare i vari argomenti passati dalla linea di comando, dobbiamo elaborare questi due membri della struttura **PSP**; questo e' proprio il lavoro svolto dal modulo **C0.ASM** del **Borland C++ 3.1**.
Vediamo un semplice esempio che visualizza il membro **CommandLineParameters** attraverso la procedura **writeString** della libreria **EXELIB**; e' necessario ricordare che appena un programma viene caricato in memoria, i due registri **DS** e **ES** contengono la componente **seg** dell'indirizzo iniziale del blocco **PSP**, mentre la componente **offset** e' implicitamente **0000h** in quanto il **PSP** e' sempre allineato al paragrafo.

```
; file argpsp.asm

;########### direttive per l'assembler ############

.386                            ; set di istruzioni a 32 bit

INCLUDE     EXELIB.INC          ; libreria di I/O

;######### dichiarazione tipi e costanti ##########

STACK_SIZE = 0400h              ; 1024 byte per lo stack

;################ segmento dati ##################

DATASEGM    SEGMENT  PARA PUBLIC USE16 'DATA'

argString   db       128 dup (0)

DATASEGM    ENDS

;############### segmento codice #################

CODESEGM    SEGMENT  PARA PUBLIC USE16 'CODE'

   assume   cs: CODESEGM        ; assegna CODESEGM a CS

start:                          ; entry point

   mov      ax, seg argString
   mov      es, ax
   mov      di, offset argString    ; es:di punta a argString
   mov      si, 81h                 ; ds:si punta a PSP:81h
   movzx    cx, byte ptr ds:[80h]   ; num. byte da trasferire
   cld                              ; DF = 0 (incremento)
   rep      movsb                   ; copia ds:si in es:di

   mov      ax, DATASEGM            ; trasferisce DATASEGM
```

```
   mov      ds, ax                  ; in DS attraverso AX
   assume   ds: DATASEGM            ; assegna DATASEGM a DS

   mov      dx, 1000h               ; riga 10h, colonna 00h
   mov      bx, offset argString    ; ds:bx punta a argString
   call     writeString

   mov      al, 00h                 ; codice di uscita
   mov      ah, 4ch                 ; proc. return to DOS
   int      21h                     ; chiama i servizi DOS

CODESEGM    ENDS

;################# segmento stack #################

STACKSEGM   SEGMENT  PARA STACK USE16 'STACK'

            db       STACK_SIZE dup (?)

STACKSEGM   ENDS

;###################################################

   END      start
```

Per copiare in **argString** il contenuto del membro **CommandLineParameters**, utilizziamo l'istruzione **MOVSB** con il prefisso **REP**; nel contatore **CX** carichiamo quindi il contenuto di **CommandLineParmLength** che rappresenta il numero di byte da copiare. Siccome **MOVSB** utilizza obbligatoriamente **ES:DI** come operando **DEST**, dobbiamo necessariamente servirci di **DS:SI** per accedere al **PSP**; queste istruzioni devono trovarsi quindi prima dell'inizializzazione di **DS** con **DATASEGM**. E' importante tener presente che in fase di esecuzione di un programma, il **DOS** svolge una serie di operazioni interne che possono anche sovrascrivere i due membri **CommandLineParmLength** e **CommandLineParameters** del **PSP**; se si ha la necessita' di elaborare questi due membri, si consiglia di salvare il loro contenuto proprio all'inizio del programma.

29.8 Assembly inline

I compilatori **C** piu' sofisticati come il **Borland C**, il **Microsoft C**, il **Watcom C**, etc, supportano una caratteristica molto potente che prende il nome di **Assembly inline**; questa caratteristica consiste nella possibilita' di inserire codice **Assembly** direttamente nel codice **C** sorgente. I compilatori **C** che supportano l'**Assembly inline**, sono dotati di appositi assemblatori incorporati; in fase di compilazione del sorgente **C**, le istruzioni scritte in **Assembly inline** vengono tradotte direttamente in codice macchina senza effettuare su di esse nessuna ottimizzazione.
La sintassi da utilizzare per la scrittura delle istruzioni **Assembly inline** varia da compilatore a compilatore; a titolo di esempio, analizziamo il caso del compilatore **Borland C++ 3.1**. Questo compilatore supporta praticamente l'intero set di istruzioni a **16** bit delle CPU **80286**, e l'intero set di istruzioni della FPU **80287**.
All'interno di un modulo **C**, una istruzione **Assembly inline** deve essere preceduta dalla parola chiave **asm**; nel caso di un blocco formato da due o piu' istruzioni, si puo' inserire il blocco stesso tra la coppia **{ }**.

Analizziamo subito un primo esempio che si riferisce al caso gia' trattato in precedenza, della somma di due strutture dati di tipo **Point3d**; il listato di questo programma e' il seguente:

```
/* file inline1.c */

#include < stdio.h >

/* tipi e costanti */

typedef struct tagPoint3d {
   int      x;
   int      y;
   int      z;
} Point3d;

/* variabili globali */

Point3d  point1 = { 1000, 1100, 1200 };
Point3d  point2 = { 2000, 2100, 2200 };

/* prototipi di funzione */

Point3d sommapoint3d(Point3d, Point3d);

/* entry point */

int main(void)
{
   Point3d  psomma;

   psomma = sommapoint3d(point1, point2);
   printf("p.x = %d, p.y = %d, p.z = %d\n",
           psomma.x, psomma.y, psomma.z);

   return 0;
}

/* sommapoint3d: somma p1 + p2 */

Point3d sommapoint3d (Point3d p1, Point3d p2)
{
   asm {
      mov      ax, word ptr p2.x
      add      word ptr p1.x, ax
      mov      ax, word ptr p2.y
      add      word ptr p1.y, ax
      mov      ax, word ptr p2.z
      add      word ptr p1.z, ax
   }
   return p1;
}
```

Da questo semplice esempio si intuisce che un programmatore **Assembly** esperto puo' fare praticamente di tutto con l'**Assembly inline**; vediamo un'altro esempio che si riferisce al caso che abbiamo gia' analizzato del calcolo della diagonale di un rettangolo:

```
/* file inline2.c */

#include < stdio.h >

/* variabili globali */

double   lato1 = 24.237865345612231;
double   lato2 = 12.654678903243517;

/* prototipi di funzione */

double diagonale(double, double);

/* entry point */

int main(void)
{
   printf("Diagonale = %.15f\n", diagonale(lato1, lato2));

   return 0;
}

/* diagonale: calcola SQRT((d1 * d1) + (d2 * d2)) */

double diagonale(double d1, double d2)
{
   double   diag;

   asm {
      finit                  /* inizializzazione FPU */
      fld      qword ptr d1   /* ST(0) = d1 */
      fmul     qword ptr d1   /* ST(0) = d1 * d1 */
      fld      qword ptr d2   /* ST(0) = d2 */
      fmul     qword ptr d2   /* ST(0) = d2 * d2 */
      fadd                   /* ST(0) = ST(0) + ST(1) */
      fsqrt                  /* ST(0) = SQRT(ST(0)) */
      fstp     qword ptr diag /* diag = ST(0) */
      fwait                  /* sincronizzazione */
   }
   return diag;
}
```

Come si puo' notare, questa volta il calcolo della diagonale viene effettuato con la **FPU** senza la necessita' di incorporare la libreria matematica del **C**; si puo' anche osservare che all'interno di un blocco **asm**, gli eventuali commenti devono essere inseriti secondo la sintassi del **C**.

Non e' consentito l'uso di etichette all'interno di un blocco **asm**; per aggirare questo problema si puo' scrivere ad esempio:

```
void init_vector(int v[], int i, int val)
{
   asm {
      push     si
      mov      bx, v
      mov      ax, val
   } init_loop:
   asm {
      mov      word ptr [bx], ax
      add      bx, 2
      dec      word ptr i
      jnz      init_loop
      pop      si
   }
}
```

E' importante notare il fatto che la funzione **init_vector** utilizza il registro **SI**; di conseguenza, il contenuto originale di **SI** deve essere preservato.
Naturalmente non si puo' pretendere che l'**Assembly inline** abbia le stesse potenzialita' dell'**Assembly** vero e prorpio; non e' possibile ad esempio utilizzare le macro, e non sono supportate diverse direttive.
Per maggiori dettagli sull'**Assembly inline** si consiglia di consultare i manuali del proprio compilatore **C**; nel caso del **Borland C++ 3.1** si possono reperire tutte le necessarie informazioni attraverso l'help in linea.

Capitolo 31 - Interfaccia tra Pascal e Assembly

In questo capitolo vengono descritte le convenzioni che permettono di interfacciare il linguaggio **Pascal** con il linguaggio **Assembly**; si presuppone che chi legge abbia una adeguata conoscenza della programmazione in **Pascal**.

Il linguaggio **Pascal** e' stato creato nel **1968** dal Prof. **Niklaus Wirth** alla **Eidgenossische Technische Hochschule** di **Zurigo**; l'obiettivo di **Wirth** era quello di creare un linguaggio di programmazione di uso generale (**general purpose**) finalizzato principalmente all'insegnamento dell'arte della programmazione ai principianti. La maggiore preoccupazione di **Wirth** fu quella di eliminare tutte le terrificanti caratteristiche che resero tristemente famoso il **BASIC**; tra gli aspetti piu' negativi del **BASIC** si possono ricordare lo scarso risalto dato alla tipizzazione dei dati e l'impossibilita' di poter creare programmi decentemente strutturati. Il risultato ottenuto da **Wirth** fu un linguaggio di programmazione che ancora oggi puo' essere definito come uno dei piu' semplici ed eleganti mai realizzati; in particolare, il **Pascal** presenta una robustissima tipizzazione dei dati e una sintassi che favorisce uno sviluppo fortemente strutturato dei programmi.
Queste caratteristiche fecero subito conquistare al **Pascal** il ruolo di linguaggio di programmazione piu' usato nelle scuole e in particolare nelle Universita' di tutto il mondo; il **Pascal** ottenne inizialmente anche un notevole successo tra i programmatori professionisti.
A differenza di quanto accade per il **C**, non esiste nessuno standard di linguaggio per il **Pascal**; proprio per questo motivo, e' possibile trovare in circolazione diverse varianti di questo linguaggio, come ad esempio l'**IBM Pascal**, il **Microsoft Quick Pascal**, il **Borland Turbo Pascal**, etc. Molto spesso queste varianti del **Pascal** possono differire tra loro per alcuni dettagli sintattici, per la presenza di estensioni del linguaggio e per il procedimento che porta alla generazione dei programmi eseguibili; in ogni caso, come molti sanno il **Borland Turbo Pascal** puo' essere definito il vero responsabile dell'enorme successo ottenuto da questo linguaggio di programmazione, tanto da meritarsi il titolo di standard "virtuale" di riferimento del linguaggio **Pascal**.
Con il passare degli anni, il **Pascal** ha subito un lento declino dovuto in particolare all'avvento del linguaggio **C**; sul finire degli anni **80** il **C** ha cominciato a prendere il sopravvento sul **Pascal** sia negli ambienti Universitari sia soprattutto tra gli sviluppatori professionisti. Questa situazione si e' verificata in quanto, come e' stato spiegato nel precedente capitolo, le rigidissime regole sintattiche del **Pascal** possono andare bene per i principianti, ma finiscono per creare notevoli problemi a quegli sviluppatori professionisti che spesso ricorrono a tecniche di programmazione piuttosto smaliziate.
La situazione comunque e' in continua evoluzione, e il **Pascal** si e' preso parzialmente una rivincita grazie al **Borland Delphi** che permette di sviluppare programmi destinati all'ambiente operativo a **32** bit di **Windows**; inoltre bisogna anche dire che le limitazioni del **Pascal** possono essere facilmente aggirate interfacciando questo linguaggio con l'**Assembly**.

30.1 Tipi di dati del Pascal

Tutte le considerazioni esposte in questo capitolo si riferiscono al compilatore **Borland Turbo Pascal** versione **7.0**; questo compilatore permette lo sviluppo di programmi **Pascal** destinati all'ambiente operativo a **16** bit del **DOS**. Come e' stato detto in precedenza, il **Turbo Pascal** viene considerato di fatto lo standard di riferimento per questo linguaggio di programmazione; l'implementazione del **Turbo Pascal** pur presentando numerose innovazioni, rispetta quasi interamente la struttura sintattica originale del linguaggio, cosi' come e' stata definita dal suo creatore **Niklaus Wirth**.

Il **Pascal** attribuisce una notevole importanza alla tipizzazione dei dati e obbliga il programmatore a tenere ben distinti quei dati che non appartengono allo stesso insieme numerico o alla stessa categoria; i tipi di dati interi forniti dal **Pascal** sono i seguenti:

Nome	Tipo	Val. Min.	Val. Max.
Shortint	intero con segno a 8 bit	-128	+127
Byte	intero senza segno a 8 bit	0	+255
Integer	intero con segno a 16 bit	-32768	+32767
Word	intero senza segno a 16 bit	0	+65535
Longint	intero con segno a 32 bit	-2147483648	+2147483647

Questi tipi di dati interi vengono chiamati **ordinali** in quanto tra due qualsiasi numeri interi puo' essere stabilita una relazione d'ordine (maggiore, minore, uguale); i tipi **ordinali** comprendono anche il tipo **Boolean** che puo' assumere uno tra i due valori costanti **False** (cioe' **0**) e **True** (cioe' **1**). I tipi **Enumerated** e **Subrange** appartengono anch'essi alla categoria dei tipi **ordinali**.

Anche il **Turbo Pascal** supporta i tre tipi fondamentali di dati in **floating point** espressi nel formato standard **IEEE**; questi tre tipi sono il **single** a **32** bit, il **double** a **64** bit e l'**extended** a **80** bit. A questi tre tipi si aggiungono anche il tipo **real** a **48** bit e il tipo **comp** a **64** bit; le caratteristiche di questi tipi di dati sono le seguenti:

Nome	Tipo	Val. Min.	Val. Max.	Precisione
single	reale con segno a 32 bit	1.5×10^{-45}	3.4×10^{38}	7-8 cifre
real	reale con segno a 48 bit	2.9×10^{-39}	1.7×10^{38}	11-12 cifre
double	reale con segno a 64 bit	5.0×10^{-324}	1.7×10^{308}	15-16 cifre
extended	reale con segno a 80 bit	3.4×10^{-4932}	1.1×10^{4932}	19-20 cifre
comp	intero con segno a 64 bit	-9.2×10^{18}	9.2×10^{18}	---

Come al solito, gli estremi **Min.** e **Max.** si riferiscono ai numeri reali positivi; per ottenere gli estremi negativi basta mettere il segno meno davanti agli estremi positivi. Il tipo **comp** e' un tipo intero con segno a **64** bit che puo' essere gestito solo attraverso la **FPU** o attraverso il software di emulazione della **FPU**.

Il **Pascal** dispone anche del tipo stringa che come gia' sappiamo, permette di definire vettori di **BYTE** dove ogni **BYTE** contiene un codice ASCII; in un precedente capitolo abbiamo anche visto che il **Pascal** utilizza il **BYTE** di indice **0** della stringa per contenere la lunghezza della stringa stessa. Possiamo dire quindi che la lunghezza massima di una stringa **Pascal** e' pari al valore massimo rappresentabile con **8** bit, e cioe' **255** caratteri.
Se scriviamo:

```
const str1[4] = 'ciao';
```

il compilatore assegna a **str1** un vettore di **5 BYTE** formato da un **BYTE** di valore **4** (lunghezza della stringa) e da **4 BYTE** contenenti i codici **ASCII** dei **4** caratteri della stringa; se invece scriviamo:

```
const str1 = 'ciao';
```

il compilatore assegna a **str1** un vettore di **256 BYTE** formato da un **BYTE** di valore **4** (lunghezza della stringa), da **4 BYTE** contenenti i codici **ASCII** dei **4** caratteri della stringa e da altri **251 BYTE** vuoti (che in genere contengono il valore **0**).

Il seguente programma permette di verificare in pratica le caratteristiche dei vari tipi di dati del **Pascal**:

```
{file datatyp.pas }

{$N+}

Program DataTypes;

{ variabili globali }

const

   si1:  Shortint = 127;
   by1:  Byte     = 240;
   in1:  Integer  = 12000;
   wo1:  Word     = 40000;
   lo1:  Longint  = 2000000000;

   sh1:  Single   = -1.63875639432690982391;
   re1:  Real     = -2.71828181543234567893;
   db1:  Double   = +3.23658976584456723456;
   ex1:  Extended = +0.87350125734897445787;
   cm1:  Comp     = +3824567893456469.0;

   st1:  String   = 'Stringa Pascal';

{ entry point }

begin
   WriteLn(si1 :25, ', ', SizeOf(si1) :2);
   WriteLn(by1 :25, ', ', SizeOf(by1) :2);
   WriteLn(in1 :25, ', ', SizeOf(in1) :2);
   WriteLn(wo1 :25, ', ', SizeOf(wo1) :2);
   WriteLn(lo1 :25, ', ', SizeOf(lo1) :2);
   WriteLn;
   WriteLn(sh1 :25:20, ', ', SizeOf(sh1) :2);
   WriteLn(re1 :25:20, ', ', SizeOf(re1) :2);
   WriteLn(db1 :25:20, ', ', SizeOf(db1) :2);
   WriteLn(ex1 :25:20, ', ', SizeOf(ex1) :2);
   WriteLn(cm1 :25:00, ', ', SizeOf(cm1) :2);
   WriteLn;
   WriteLn(st1, ', ', Ord(st1[0]), ', ', SizeOf(st1));
end.
```

La funzione **SizeOf** restituisce la dimensione in byte del suo argomento; la funzione **Ord** restituisce il valore numerico (**ordinale**) del suo argomento che deve appartenere ai tipi ordinali.

30.2 Convenzioni per i compilatori Pascal

Segmenti di programma

I compilatori **Pascal** capaci di generare il codice oggetto, utilizzano per i segmenti di programma le stesse convenzioni illustrate nei precedenti capitoli; la possibilita' di avere a disposizione il codice oggetto di un modulo **Pascal** facilita notevolmente il lavoro di interfacciamento con l'**Assembly**.
Il compilatore **Turbo Pascal** invece produce un codice intermedio chiamato **P-Code**, che viene tenuto nascosto al programmatore; questa situazione rende piu' difficoltose le comunicazioni tra moduli **Turbo Pascal** e moduli **Assembly**. Si tenga presente inoltre che il **Turbo Pascal** utilizza una segmentazione che segue solo in parte le convenzioni che gia' conosciamo; per rendersene conto basta richiedere al compilatore la generazione del **map file** del programma. A tale proposito bisogna selezionare il menu **Options - Linker - Map File**; attraverso il **map file** si puo' constatare quanto segue:

Esiste un unico blocco dati comprendente quindi i dati inizializzati, non inizializzati e costanti; questo blocco presenta le seguenti caratteristiche:
```
DATA SEGMENT PARA PUBLIC USE16 'DATA'
```

Esiste un unico blocco stack che naturalmente viene predisposto dal compilatore, e presenta le seguenti carattesristiche:
```
STACK SEGMENT PARA STACK USE16 'STACK'
```

Ciascun modulo appartenente ad un programma **Pascal** utilizza un proprio segmento di codice; i vari segmenti di codice differiscono tra loro solo per il nome. Il segmento di codice relativo al modulo principale di un programma **Pascal** contiene l'**entry point** e quindi ricopre il ruolo di segmento principale di codice; il nome utilizzato da questo segmento e' legato alla presenza o meno della parola riservata **Program** all'inizio del modulo principale. Se non utilizziamo questa parola riservata, il blocco di codice principale assume le seguenti caratteristiche:
```
PROGRAM SEGMENT PARA PUBLIC USE16 'CODE'
```
Se, come nel precedente esempio **DATATYP.PAS**, il programma inizia con:
```
Program DataTypes;
```
allora il blocco di codice principale assume le seguenti caratteristiche:
```
DataTypes SEGMENT PARA PUBLIC USE16 'CODE'
```

Ogni **unit** del **Turbo Pascal** utilizza un segmento di codice che ha il nome della **unit** stessa; nel caso ad esempio della **unit Crt** della libreria **Pascal**, viene utilizzato il seguente segmento di codice:
```
Crt SEGMENT PARA PUBLIC USE16 'CODE'
```

Da queste considerazioni emerge chiaramente il fatto che un programma **Turbo Pascal** e' dotato di un unico segmento di dati, un unico segmento di stack e due o piu' segmenti di codice; le procedure presenti nelle **unit** si trovano in segmenti di codice diversi da quello in cui si trova il caller, e quindi possono essere chiamate solo attraverso **FAR** calls. Nel caso dei moduli **Pascal** tutti questi aspetti vengono gestiti dal compilatore; nel caso dei moduli **Assembly** invece tutto e' nelle mani del programmatore. In particolare, il programmatore deve indicare chiaramente al compilatore **Pascal** il tipo **NEAR** o **FAR** delle procedure definite nei moduli **Assembly**; il procedimento da seguire viene illustrato nel seguito del capitolo.

Al momento di caricare il programma in memoria, il **SO** pone **SS=STACK**, mentre **CS** viene inizializzato con il segmento principale di codice che contiene ovviamente l'**entry point**; nel caso del precedente esempio **DATATYP.PAS**, si ha **CS=DataTypes**. Da parte sua il compilatore genera il codice macchina che pone automaticamente **DS=DATA**; esistendo un unico blocco di dati, non viene creato nessun gruppo **DGROUP**.

Stack frame

Il **Pascal** mette a disposizione due tipi di procedure definibili attraverso le parole riservate **procedure** e **function**; a differenza di quanto accade con una **procedure**, la **function** ha sempre un valore di ritorno. Nel seguito del capitolo viene utilizzato il termine procedura per indicare genericamente sia una **procedure** che una **function**.
La convenzione **Pascal** prevede che gli eventuali argomenti da passare ad una procedura vengano inseriti nello stack a partire dal primo (sinistra destra); all'interno dello stack gli argomenti di una procedura vengono quindi posizionati in ordine inverso rispetto a quello indicato dalla intestazione della procedura stessa. Il compito di ripulire lo stack dagli argomenti spetta come sappiamo alla procedura chiamata; questo lavoro viene generalmente effettuato attraverso l'istruzione:

```
RET n
```

dove **n** indica il numero di byte da sommare a **SP**.
Come al solito, l'accesso ai parametri di una procedura avviene attraverso **BP**; questa volta pero' bisogna ricordarsi che i parametri sono posizionati in ordine inverso nello stack, e quindi muovendoci in avanti rispetto a **BP** incontreremo per primo l'ultimo parametro ricevuto dalla procedura. Nel caso di **NEAR** call l'ultimo parametro si trova a **BP+4**; nel caso invece di **FAR** call l'ultimo parametro si trova a **BP+6**.

Valori di ritorno

In relazione alle locazioni utilizzate per contenere gli eventuali valori di ritorno delle **function**, valgono tutte le convenzioni illustrate nel **Capitolo 24**; anche nel **Turbo Pascal** quindi abbiamo la seguente situazione:
***** Gli interi a **8** bit vengono restituiti negli **8** bit meno significativi di **AX**, e cioe' in **AL**.
***** Gli interi a **16** bit (compresi gli offset **NEAR**) vengono restituiti in **AX**.
***** Gli interi a **32** bit vengono restituiti nella coppia **DX:AX**, con **DX** che in base alla notazione **little-endian** contiene la **WORD** piu' significativa.
***** Gli indirizzi **FAR** a **16+16** bit vengono restituiti sempre nella coppia **DX:AX**, con **DX** che contiene la componente **seg**, e **AX** la componente **offset**.
***** Tutti i valori in virgola mobile **IEEE** di tipo **single**, **real**, **float** e **extended** vengono restituiti nel registro **ST(0)** della **FPU**; lo stesso discorso vale per il tipo speciale **comp**.

30.3 Le classi di memoria del Pascal

In relazione alle classi di memoria, bisogna premettere che nel caso piu' generale, un programma **Pascal** e' formato da un modulo principale **Pascal**, da una o piu' **unit Pascal** e da uno o piu' moduli **Assembly**; si tenga presente che nel caso del **Turbo Pascal**, la **unit System** contenente le procedure di sistema (come **New**, **Dispose**, **Write**, **Sin**, **Cos**, etc), viene incorporata automaticamente nel modulo principale e in tutte le **unit** del programma. Osservando infatti il **map file** di un qualunque programma **Turbo Pascal**, si nota sempre la presenza di un blocco codice:

```
System SEGMENT PARA PUBLIC USE16 'CODE'
```

Il **Pascal** permette di creare procedure innestate all'interno di altre procedure; una procedura **A** innestata in una procedura **B** e' visibile solamente in **B** e quindi non puo' essere chiamata da altre procedure. In sostanza, i compilatori **Pascal** permettono di trattare le procedure innestate come se fossero delle variabili locali create nello stack; e' ovvio pero' che in realta', una qualunque procedura innestata o non innestata, viene sempre creata in un segmento di codice del programma.
Tutte le procedure non innestate presenti nel modulo principale **Pascal** hanno automaticamente linkaggio esterno; queste procedure quindi sono visibili anche negli eventuali moduli **Assembly**

facenti parte del programma. In relazione alle **unit** il discorso e' analogo al caso delle procedure **static** e non **static** del **C**; in sostanza, la distinzione tra procedure pubbliche e private di una **unit** viene gestita dal compilatore, e quindi vale solamente per i moduli **Pascal** che fanno uso della **unit** stessa. Un modulo **Assembly** invece e' anche in grado di vedere tutte le procedure non innestate presenti nella sezione **implementation** di una **unit**; se vogliamo attenerci alle regole di visibilita' del **Pascal**, nei moduli **Assembly** possiamo evitare di dichiarare **EXTRN** le procedure private di una **unit**.
Come e' stato detto in precedenza, l'eccezione e' rappresentata dalla **unit System**; questa **unit** speciale viene automaticamente collegata al modulo principale e a tutte le altre **unit** attraverso un procedimento che viene nascosto al programmatore (si puo' anche constatare che non esiste nessun file **SYSTEM.TPU**). La conseguenza e' che le procedure della **unit System** non risultano visibili nei moduli **Assembly**; nei moduli **Pascal** inoltre e' proibito passare queste procedure come argomenti di altre procedure.
Gli identificatori delle procedure innestate presenti in un qualunque modulo **Pascal** hanno linkaggio interno e non risultano visibili negli altri moduli, compresi i moduli **Assembly**.

Le etichette dichiarate con **label** in una procedura sono visibili solo all'interno della procedura stessa, e il loro identificatore puo' essere quindi ridefinito in altri punti del programma; le etichette dichiarate con **label** al di fuori di qualsiasi procedura sono visibili in tutto il modulo di appartenenza, e il loro identificatore puo' essere quindi ridefinito in altri moduli.

Per quanto riguarda le variabili globali dei moduli **Pascal**, e cioe' le variabili definite al di fuori di qualsiasi procedura, valgono le stesse regole esposte per le procedure; tutte le variabili definite al di fuori di qualsiasi procedura del modulo principale **Pascal**, hanno automaticamente linkaggio esterno e risultano quindi visibili anche negli eventuali moduli **Assembly** facenti parte del programma. In realazione alle **unit**, la distinzione tra variabili globali pubbliche e private viene gestita dal compilatore, e quindi vale solamente per i moduli **Pascal** che fanno uso della **unit** stessa; in pratica, un modulo **Pascal** che fa uso di una **unit**, e' in grado di vedere solamente le variabili globali presenti nella sezione **interface** della **unit** stessa. Un modulo **Assembly** invece e' in grado di vedere tutte le variabili globali presenti sia nella sezione **interface**, sia nella sezione **implementation** di una **unit**; anche in questo caso, se vogliamo attenerci alle regole di visibilita' del **Pascal**, nei moduli **Assembly** possiamo evitare di dichiarare **EXTRN** le variabili globali private di una **unit**. I moduli **Assembly** non sono invece in grado di vedere le costanti prive di tipo dichiarate in un modulo **Pascal**, come ad esempio:

```
const ALTEZZA = 10;
```

Come e' stato spiegato in precedenza, tutte le variabili globali di un programma **Pascal** (pubbliche o private), vengono sistemate in un unico segmento dati definito come:

```
DATA SEGMENT PARA PUBLIC USE16 'DATA'
```

Il compilatore assegna automaticamente il valore **0** a tutte le variabili globali non inizializzate; analogamente, le stringhe non inizializzate vengono riempite con zeri.
Tutte le variabili definite in una procedura **Pascal** sono considerate **variabili locali**; le variabili locali vengono create nello stack e risultano visibili solo all'interno della procedura di appartenenza. Le variabili locali possono essere inizializzate sia con valori costanti che con il contenuto di altre variabili; come al solito, in assenza di inizializzazione il contenuto di una variabile locale e' casuale.

30.4 Gestione degli indirizzamenti in Pascal

Anche il **Pascal** supporta i puntatori che come sappiamo sono delle variabili intere senza segno il cui contenuto rappresenta un indirizzo di memoria; il problema che si presenta e' dato dal fatto che il **Pascal** e' un linguaggio di programmazione destinato ai principianti, per cui tende a nascondere tutti i dettagli relativi alla gestione a basso livello di un programma. La conseguenza pratica e' che in **Pascal** la gestione dei puntatori e' piuttosto contorta e tende a creare parecchi problemi ai programmatori provenienti dal **C** o dall'**Assembly**; il modo migliore per aggirare questi problemi consiste naturalmente nel ricorrere in caso di necessita' all'interfacciamento con l'**Assembly** o all'uso dell'**Assembly inline** supportato in modo molto efficiente dal **Turbo Pascal**.
La prima cosa da dire riguarda il fatto che nel **Turbo Pascal** i puntatori sono sempre di tipo **FAR**, e rappresentano quindi una coppia **seg:offset** da **16+16** bit; come al solito, in base alla convenzione **Intel** la componente **seg** deve sempre trovarsi nei **16** bit piu' significativi del puntatore. In base a questa premessa, tutti gli aspetti relativi ai puntatori del **Pascal** coincidono con le cose dette nel precedente capitolo a proposito della gestione dei puntatori con il **C**; per maggiore chiarezza, consideriamo il seguente blocco di variabili globali di un modulo principale **Pascal**:

```
type
   ptShortint  = ^Shortint;   { tipo puntatore a Shortint }
   ptInteger   = ^Integer;    { tipo puntatore a Integer }
   ptLongint   = ^Longint;    { tipo puntatore a Longint }
   ptString    = ^String;     { tipo puntatore a String }

const
   c1:   Shortint = 12;       { intero con segno a 8 bit }
   i1:   Integer  = 1350;     { intero con segno a 16 bit }
   l1:   Longint  = 186900;   { intero con segno a 32 bit }
   s1:   String   = 'Stringa Pascal';  { stringa da 256 byte }

var
   ptSho:   ptShortint;       { variabile puntatore a Shortint }
   ptInt:   ptInteger;        { variabile puntatore a Integer }
   ptLon:   ptLongint;        { variabile puntatore a Longint }
   ptStr:   ptString;         { variabile puntatore a String }
```

Quando un compilatore **Pascal** incontra questo blocco dati:

***** assegna a **c1** una locazione di memoria da **8** bit, e carica in questa locazione il valore **12**;
***** assegna a **i1** una locazione di memoria da **16** bit, e carica in questa locazione il valore **1350**;
***** assegna a **l1** una locazione di memoria da **32** bit, e carica in questa locazione il valore **186900**;
***** assegna a **s1 256** locazioni di memoria da **8** bit ciascuna, e carica nella prima locazione il valore **14** (lunghezza della stringa), nelle successive **14** locazioni i codici **ASCII** dei **14** caratteri della stringa, e nelle restanti **241** locazioni il valore **0**;
***** assegna a **ptSho** una locazione di memoria da **16+16** bit, e carica in questa locazione la coppia **0000h:0000h** (**nil**);
***** assegna a **ptInt** una locazione di memoria da **16+16** bit, e carica in questa locazione la coppia **0000h:0000h** (**nil**);
***** assegna a **ptLon** una locazione di memoria da **16+16** bit, e carica in questa locazione la coppia **0000h:0000h** (**nil**);
***** assegna a **ptStr** una locazione di memoria da **16+16** bit, e carica in questa locazione la coppia **0000h:0000h** (**nil**).

All'interno del blocco di codice principale, cioe' all'interno del blocco delimitato da **begin** e **end**, procediamo alla inizializzazione dei puntatori; in presenza dell'istruzione:

```
ptSho^:= c1;
```

il compilatore carica in **ptSho** l'indirizzo **seg:offset** di **c1**.
In presenza dell'istruzione:
```
ptInt^:= i1;
```
il compilatore carica in **ptInt** l'indirizzo **seg:offset** di **i1**.
In presenza dell'istruzione:
```
ptLon^:= l1;
```
il compilatore carica in **ptLon** l'indirizzo **seg:offset** di **l1**.
In presenza dell'istruzione:
```
ptStr^:= s1;
```
il compilatore carica in **ptStr** l'indirizzo **seg:offset** del primo elemento di **s1**.

Come si puo' notare, la sintassi utilizzata dal **Pascal** con i puntatori lascia molto a desiderare e tende spesso a creare una certa confusione; molti programmatori provenienti dal **C** ad esempio, seguendo la logica sono portati a scrivere:
```
ptInt:= ^i1;
```
Nel tentativo di rendere meno confusa la gestione dei puntatori, il **Turbo Pascal** ha introdotto un nuovo operatore rappresentato dal simbolo **@** (chiocciola); questo operatore equivale al simbolo **&** del **C** e deve essere letto come **indirizzo di**. Impiegando questo nuovo operatore possiamo scrivere assegnamenti del tipo:
```
ptInt:= @i1;
```
Come si puo' notare, questa nuova sintassi appare molto piu' chiara e logica di quella utilizzata dal **Pascal** classico; la precedente istruzione infatti indica chiaramente che stiamo assegnando a **ptInt** l'indirizzo di **i1**.
Il **Turbo Pascal** mette a disposizione anche i puntatori generici rappresentati dal tipo **Pointer**; questi puntatori equivalgono ai puntatori a **void** del **C**. Un puntatore di tipo **Pointer** puo' essere fatto puntare a qualsiasi variabile di qualsiasi tipo; nella sezione **var** del blocco dati illustrato in precedenza, possiamo scrivere ad esempio:
```
ptVoid: Pointer;
```
A questo punto, nel blocco di codice principale possiamo scrivere:
```
ptVoid:= @c1;
```
In qualsiasi altro punto del programma possiamo anche far puntare **ptVoid** altrove; possiamo scrivere ad esempio:
```
ptVoid:= @ptStr;
```
Come accade per i puntatori a **void** del **C**, anche i **Pointer** del **Turbo Pascal** non possono essere dereferenziati; il loro scopo e' solamente quello di memorizzare un indirizzo che spesso viene poi passato ad una procedura **Assembly** che puo' gestirlo senza avere tra i piedi lo stretto controllo di tipo dei compilatori **Pascal**.
Il **Turbo Pascal** mette a disposizione anche altre procedure di basso livello per i puntatori; in particolare, possono tornale utili le **function** come **Seg**, **Ofs**, **Ptr** e **Addr**.
Seg richiede un solo argomento che deve essere l'identificatore pubblico di una variabile, di una **function** o di una **procedure**; il valore restituito da **Seg** e' una **Word** contenente la componente **seg** dell'argomento.
Ofs richiede un solo argomento che deve essere l'identificatore pubblico di una variabile, di una **function** o di una **procedure**; il valore restituito da **Ofs** e' una **Word** contenente la componente **offset** dell'argomento.
Addr richiede un solo argomento che deve essere l'identificatore pubblico di una variabile, di una **function** o di una **procedure**; il valore restituito da **Addr** e' un **Pointer** contenente la coppia **seg:offset** dell'argomento.
Ptr richiede due argomenti di tipo **Word** che devono rappresentare una coppia **seg:offset**; il valore restituito da **Ptr** e' un **Pointer** contenente la coppia **seg:offset** costituita dalle due **Word** passate come argomenti.

Tutte queste estensioni del **Pascal** sono state introdotte con l'intento di rendere il linguaggio piu'

potente e piu' flessibile; molto spesso pero' si ottiene come risultato un notevole aumento della confusione. Consideriamo ad esempio la seguente procedura **Pascal** che si serve del blocco dati che abbiamo visto in precedenza:

```
procedure WriteInteger(var i: Integer);
begin
   WriteLn(i);
end;
```

Questa procedura richiede un **Integer** passato per indirizzo; se vogliamo passare a questa procedura la variabile **i1** per indirizzo, dobbiamo scrivere:
`WriteInteger(i1);`
Il **Pascal** come al solito tende a nascondere tutti i dettagli relativi alla gestione a basso livello del programma; la variabile **i1** ci appare quindi come se venisse passata per valore.
Un programmatore abituato a lavorare con il **C**, dopo aver fatto puntare **ptInt** a **i1** cercherebbe di scrivere:
`WriteInteger(ptInt);`
Il compilatore genera pero' un messaggio di errore per indicare che **WriteInteger** richiede un **Integer** e non un puntatore ad **Integer**; se vogliamo utilizzare per forza **ptInt** dobbiamo scrivere allora:
`WriteInteger(ptInt^);`
Secondo la sintassi del **Pascal** infatti, **ptInt^** e' la variabile puntata da **ptInt** e cioe' **i1** che e' un **Integer**; queste assurdita' sono legate in parte agli stretti vincoli sulla compatibilita' dei tipi di dati del **Pascal**, e in parte al fatto che il linguaggio **Pascal** e' stato progettato proprio per impedire ai programmatori di ricorrere a questi "giochetti".
La procedura **WriteInteger** puo' essere resa piu' flessibile in questo modo:

```
procedure WriteInteger(pi: ptInteger);
begin
   WriteLn(pi^);
end;
```

In questo caso, supponendo che **ptInt** stia puntando a **i1**, possiamo effettuare la seguente chiamata:
`WriteInteger(ptInt);` (passaggio indiretto dell'indirizzo di **i1**).
Alternativamente, grazie alle estensioni del **Turbo Pascal** si puo' anche scrivere:
`WriteInteger(@i1);` (passaggio diretto dell'indirizzo di **i1**),
oppure:
`WriteInteger(Addr(i1));` (passaggio diretto dell'indirizzo di **i1**).
Appare chiaro pero' che per poter scrivere codice **Pascal** di questo genere e' necessario avere una adeguata conoscenza dell'**Assembly**; in altre parole, il programmatore deve conoscere tutti i dettagli sul funzionamento a basso livello di un programma **Pascal**.
Come e' stato detto in precedenza, per aggirare tutti questi problemi conviene spesso ricorrere all'interfacciamento con l'**Assembly** in modo da avere a disposizione una sintassi molto piu' chiara e semplice soprattutto per i puntatori; come vedremo nel seguito del capitolo, si rivela estremamente efficace anche il ricorso all'**Assembly inline**.

Una procedura **Assembly** che riceve come argomento un puntatore **Pascal**, si trova ad avere a che fare con una normalissima coppia **seg:offset**; a questo punto, la gestione di questa coppia **seg:offset** si svolge nello stesso identico modo gia' illustrato nei precedenti capitoli. E' importante solo

ricordare che tutti i puntatori del **Turbo Pascal** sono di tipo **FAR**; di conseguenza, una procedura **Assembly** che restituisce un puntatore ad un modulo **Pascal**, deve restituire la coppia completa **seg:offset**.

Un'ultima considerazione riguarda il fatto che anche il **Pascal** permette di passare una procedura **A** come argomento di un'altra procedura **B**; come abbiamo visto nei precedenti capitoli, in un caso del genere viene passato come argomento l'indirizzo di memoria da cui inizia il corpo della procedura **A**, e cioe' l'indirizzo della prima istruzione della procedura **A**.
Per gestire questa situazione, la sintassi utilizzata dal **Turbo Pascal** si discosta da quella definita nel **Pascal** classico; vediamo a tale proposito un esempio pratico. Suppponiamo di voler scrivere una procedura che riceve come argomento un'altra procedura avente le caratteristiche della **WriteInteger** vista in precedenza; prima di tutto dobbiamo creare un apposito tipo di dato, e quindi nella sezione **type** del programma possiamo scrivere ad esempio:

```
ptProcInt = procedure(var i: Integer);
```

In questo modo abbiamo dichiarato un nuovo tipo di dato **ptProcInt** che rappresenta un tipo puntatore a una **procedure** che richiede un argomento di tipo **Integer** da passare per indirizzo; a questo punto possiamo creare la nostra procedura che avra' una intestazione del tipo:

```
procedure ProcTest(pp1: ptProcInt, var i: Integer);
```

Nel blocco delle istruzioni principali possiamo ora effettuare chiamate del tipo:

```
ProcTest(WriteInteger, i1);
```

In un caso del genere quindi, la procedura **ProcTest** riceve come primo argomento la coppia **seg:offset** che rappresenta l'indirizzo di memoria da cui inizia il corpo di **WriteInteger**; come secondo argomento **ProcTest** riceve in modo "occulto" la coppia **seg:offset** di **i1** e non il valore di **i1**.

30.5 Protocollo di comunicazione tra Pascal e Assembly

Analizziamo ora le regole che permettono a due o piu' moduli **Pascal** e **Assembly** di comunicare tra loro; l'insieme di queste regole definisce il protocollo di comunicazione tra **Pascal** e **Assembly**.

Un modulo **Assembly** che fa parte di un programma **Pascal**, puo' definire le sue eventuali variabili globali in uno o piu' blocchi di dati che possono avere caratteristiche scelte a piacere dal programmatore; questi blocchi infatti vengono totalmente ignorati dal compilatore **Pascal**. La conseguenza pratica di tutto cio' e' data dal fatto che le variabili globali presenti in un modulo **Assembly** risultano invisibili nei moduli **Pascal**; se proviamo a dichiarare **PUBLIC** queste variabili, otteniamo un messaggio di errore da parte del compilatore **Pascal**.
Un modulo **Assembly** che fa parte di un programma **Pascal**, puo' definire le sue procedure in uno o piu' blocchi di codice che possono avere caratteristiche scelte a piacere dal programmatore; anche in questo caso pero', questi blocchi di codice vengono ignorati dal compilatore **Pascal** e non possono dichiarare procedure **PUBLIC**. Se vogliamo che le procedure presenti in un modulo **Assembly** risultino visibili anche nel modulo principale **Pascal** o nelle **unit Pascal**, dobbiamo inserire le definizioni di queste procedure in un blocco di codice che deve chiamarsi obbligatoriamente **CODE**; in generale, le caratteristiche di questo segmento di codice sono le seguenti:

```
CODE SEGMENT PARA PUBLIC USE16 'CODE'
```

Tutte le procedure dichiarate **PUBLIC** in questo blocco di codice, risultano visibili anche nei moduli **Pascal**; alternativamente e' possibile definire questo blocco di codice anche in questo modo:

```
_TEXT SEGMENT PARA PUBLIC USE16 'CODE'
```

Questo significa che se vogliamo utilizzare le carattersitiche avanzate di **MASM** e **TSASM**, possiamo servici della direttiva semplificata **.CODE**; in ogni caso si tenga presente che e' fondamentale l'utilizzo del nome **CODE** o **_TEXT**. Queste strane regole sono una diretta

conseguenza del fatto che il compilatore **Turbo Pascal** non genera nessun **object file**; questo fatto ci obbliga ad effettuare il linking di un modulo **Assembly** secondo uno schema imposto dal compilatore stesso.

Il **Turbo Pascal** utilizza due soli modelli di memoria che equivalgono al modello **SMALL** e al modello **LARGE**; per abilitare il modello **LARGE** e' necessario selezionare il menu **Options - Compiler - Force far calls**. Alternativamente e' anche possibile inserire la direttiva **{$F+}** all'inizio di tutti i moduli **Pascal**; analogamente, la direttiva **{$F-}** disabilita le **FAR** calls.
Se scrivamo un programma **Pascal** che fa un uso massiccio dei puntatori, possiamo imbatterci in messaggi di errore del tipo:

```
Invalid procedure or function reference
```

Questo messaggio di errore e' legato spesso al fatto che abbiamo disabilitato le **FAR** calls; per evitare qualsiasi problema, si raccomanda vivamente quindi di abilitare sempre le **FAR** calls, soprattutto quando si interfaccia il **Pascal** con l'**Assembly**. Di conseguenza, e' importantissimo che tutte le procedure **PUBLIC** presenti in un modulo **Assembly** siano di tipo **FAR**; se stiamo utilizzando le caratteristiche avanzate di **MASM** e **TASM** dobbiamo servirci necessariamente della direttiva:

```
.MODEL LARGE, PASCAL
```

I moduli **Pascal** che intendono utilizzare le procedure **PUBLIC** definite nel blocco **CODE** di un modulo **Assembly**, devono dichiarare queste procedure con il qualificatore **external**; la sintassi del **Turbo Pascal** prevede inoltre che venga utilizzata la direttiva **{$L nomefile.obj}** per indicare in quale **object file** si trova la procedura **Assembly** esterna. Per ogni **object file** e' necessaria una sola direttiva **{$L}**; l'**object file** deve essere in formato **OMF** (Object Module Format).
Supponiamo ad esempio di aver creato in un modulo **ASMMOD.ASM** una procedura **PUBLIC** chiamata **AreaCerchio**, ed avente il seguente prototipo **Pascal**:

```
function AreaCerchio(r: double): double;
```

Nel modulo **Pascal** che intende utilizzare questa **function** dobbiamo prima di tutto inserire la direttiva:

```
{$L asmmod.obj}
```

A questo punto nello stesso modulo **Pascal** dobbiamo dichiarare **AreaCerchio** come:

```
function AreaCerchio(r: double): double; external;
```

Nel caso di una **unit Pascal**, la precedente dichiarazione deve trovarsi nella sezione **implementation**; se vogliamo che **AreaCerchio** sia visibile anche all'esterno della **unit** dobbiamo inserire nella sezione **interface** la dichiarazione:

```
function AreaCerchio(r: double): double;
```

Come si puo' notare, la dichiarazione inserita nella sezione **interface** e' priva del qualificatore **external**.

Tutte le procedure contenute nei moduli **Assembly** da linkare al **Pascal**, devono preservare rigorosamente il contenuto dei registri **CS**, **DS**, **SS**, **SP** e **BP**; se si utilizzano le caratteristiche avanzate di **MASM** e **TASM**, all'interno delle procedure dotate di **stack frame** il contenuto dei registri **SP** e **BP** viene automaticamente preservato dall'assembler. Tutti gli altri registri sono a completa disposizione del programmatore; per questi registri quindi non e' necessario preservarne il contenuto.

Il **Pascal** e' un linguaggio **case-insensitive**, e quindi i compilatori **Pascal** non distinguono tra maiuscole e minuscole; possiamo dire quindi che in **Pascal** i nomi come **varword1**, **VarWord1**, **VARWORD1**, etc, rappresentano tutti lo stesso identificatore. Se in un modulo **Pascal** definiamo una variabile chiamata **varword1**, e poi proviamo a definire una seconda variabile chiamata

VARWORD1, otteniamo un messaggio di errore da parte del compilatore; in osservanza a queste regole, al momento di assemblare un modulo **Assembly** da linkare ad un modulo **Pascal**, non dobbiamo assolutamente utilizzare le opzioni come **/ml** per il **TASM** o **/Cp** per il **MASM**.

30.6 Esempi pratici

In base a tutte le considerazioni appena esposte, e in base alle cose dette nei precedenti capitoli, possiamo facilmente scrivere programmi di qualunque complessita', formati da un numero arbitrario di moduli **Pascal** e moduli **Assembly**; vediamo allora un unico esempio formato complessivamente da **4** moduli distinti chiamati **PASMOD.PAS**, **ASMMODA.ASM**, **UNITMOD.PAS** e **ASMMODB.ASM**.
Il modulo **PASMOD.PAS** ricopre il ruolo di modulo principale del programma, e contiene quindi anche l'**entry point** e il blocco **begin end** che rappresenta il blocco di codice principale; inoltre il modulo **PASMOD.PAS** si serve delle procedure definite nel modulo **ASMMODA.ASM**. Il modulo **UNITMOD.PAS** e' una **unit** contenente una serie di procedure utilizzate anch'esse da **PASMOD.PAS**; la **unit UNITMOD.PAS** a sua volta si serve delle procedure definite nel modulo **ASMMODB.ASM**.
Il modulo principale **PASMOD.PAS** assume il seguente aspetto:

```
{ file pasmod.pas }

{ direttive per il compilatore }

{$F+}
{$L asmmoda.obj}

{ intestazione del programma }

Program PascalModule;

{ inclusione units }

uses

   Crt, UnitMod;

{ dichiarazione tipi di dati }

type

   vStrType    = Array[0..9] of String[8];

   recCliente  = record
      codice:     Integer;
      telefono:   Longint;
      nome:       String[42];
   end;

{ variabili globali inizializzate }

const

   codCli1: Integer     = 10201;
   telCli1: Longint     = 871111111;
   strCli1: String[64]  = 'Mario Rossi';

   raggio:  Double      = 10.456723594562316;
```

```
   strPas:  String = 'ESEMPIO DI INTERFACCIA TRA PASCAL E ASSEMBLY';

   vColors: vStrType = (
      'Rosso', 'Verde', 'Giallo', 'Bianco', 'Marrone',
      'Grigio', 'Blu', 'Nero', 'Rosa', 'Azzurro'
   );

{ variabili globali non inizializzate }

var

   recCli1: recCliente;
   recCli2: recCliente;

   vCol:    vStrType;
   i:       Integer;

{ dichiarazione procedure definite in asmmoda.asm }

procedure InitScreen(bgColor, fgColor: Byte); external;
procedure InitRecCli(prc1, prc2: Pointer); external;

{ implementazione procedure globali di pasmod.pas }

procedure WriteString(var s: String; d: Word);
begin
   WriteLn(s :d);
end;

{ entry point del programma }

begin
   InitScreen(Blue, Yellow);
   WriteLn;

   recCli2.codice:= codCli1;
   recCli2.telefono:= telCli1;
   recCli2.nome:= strCli1;

   InitRecCli(@recCli1, @recCli2);

   WriteLn('Cliente1:');
   WriteLn('Codice:   ', recCli.codice :20);
   WriteLn('Telefono: ', recCli1.telefono :20);
   WriteLn('Nome:     ', recCli1.nome :20);
   WriteLn;

   UnitProcedure(@vCol, @vColors);
   for i:=0 to 9 do
      WriteLn('vCol[', i, '] = ', vCol[i]);
   WriteLn;

   Writeln('Raggio =       ', raggio :25:20);
   WriteLn('Area Cerchio = ', AreaCerchio(raggio) :25:20);
end.
```

Prima di tutto notiamo la direttiva **{$F+}** che abilita le **FAR** calls per tutte le procedure di **PASMOD.PAS**; e' presente anche una direttiva **{$L}** che richiede il linking di **PASMOD.PAS** con il modulo **ASMMODA.OBJ**.
Il modulo **PASMOD.PAS** contiene l'intestazione **PascalModule**; possiamo dire quindi che tutto il

codice di questo modulo verra' inserito in un segmento di programma avente le seguenti caratteristiche:

```
PascalModule SEGMENT PARA PUBLIC USE16 'CODE'
```

Nella sezione **uses** vengono incluse la **unit Crt** e la **unit UnitMod**; come vedremo in seguito, l'inclusione di **Crt** e' necessaria per permettere anche al modulo **ASMMODA.ASM** di utilizzare le procedure definite in questa **unit**.
Nella sezione **type** viene dichiarato un tipo di dato **vStrType** che rappresenta un vettore di **10** stringhe formate ciascuna da **8 BYTE**; il compilatore come sappiamo assegna in realta' **9 BYTE** ad ogni stringa in quanto il primo **BYTE** e' destinato a contenere la lunghezza della stringa stessa.
Complessivamente quindi ogni variabile di tipo **vStrType** richiede:
`10 * 9 = 90` byte di memoria.
Successivamente viene dichiarato un tipo di dato **recCliente** che e' un **record** formato da un campo **codice** a **16** bit, da un campo **telefono** a **32** bit e da un campo **nome** che e' una stringa da **43 BYTE** (cioe' **42 BYTE** piu' il **BYTE** iniziale contenente la lunghezza della stringa stessa); complessivamente, una variabile di tipo **recCliente** richiede:
`2 + 4 + 43 = 49` byte di memoria.
Le sezioni **const** e **var** dichiarano una serie di variabili che verranno utilizzate dal programma; come sappiamo, tutte queste variabili sono visibili anche all'esterno del modulo **PASMOD.PAS**.
Subito dopo l'**entry point**, viene chiamata una procedura **InitScreen** definita nel modulo **ASMMODA.ASM**; questa procedura richiede due parametri di tipo **Byte** che vengono utilizzati per inizializzare lo schermo con un colore di sfondo (**bgColor**) e con un colore di primo piano (**fgColor**). Le costanti predefinite come **Yellow**, **Blue**, etc, vengono create nella **unit Crt**; a tale proposito si puo' consultare il file **CRT.INT** presente nella cartella **DOC** del **Turbo Pascal**.
Il compito successivo svolto dal modulo principale consiste nell'inizializzare una variabile **recCli2** di tipo **recCliente**; questa variabile viene poi utilizzata per inizializzare un'altra variabile **recCli1** sempre di tipo **recCliente**. A tale proposito viene chiamata una procedura **InitRecCli** definita nel modulo **ASMMOD.ASM**; come si puo' notare, **InitRecCli** richiede due argomenti di tipo puntatore a **recCliente**. Il parametro **prc1** punta al **record** destinazione, mentre il parametro **prc2** punta al **record** sorgente; per dimostrare che **InitRecCli** ha veramente copiato **recCli2** in **recCli1**, vengono visualizzati tutti i campi dello stesso **recCli1**. Nella chiamata:

```
InitRecCli(@recCli1, @recCli2);
```

vengono passati direttamente gli indirizzi dei due argomenti; tutto cio' equivale a scrivere:

```
InitRecCli(Addr(recCli1), Addr(recCli2));
```

Se vogliamo passare indirettamente gli indirizzi dei due argomenti, dobbiamo servirci delle variabili puntatore; definendo due variabili **Ptr1** e **Ptr2** di tipo **Pointer**, dopo aver fatto puntare **Ptr1** a **recCli1** e **Ptr2** a **recCli2** possiamo scrivere semplicemente:

```
InitRecCli(Ptr1, Ptr2);
```

L'ultimo compito svolto dal modulo principale consiste nel chiamare le due procedure **UnitProcedure** e **AreaCerchio** definite nella **unit UnitMod**; la procedura **UnitProcedure** richiede due argomenti di tipo puntatore a **vStrType**. Il contenuto della locazione puntata dal secondo puntatore (sorgente) viene copiato nella locazione puntata dal primo puntatore (destinazione); anche in questo caso, per dimostrare che **UnitProcedure** ha effettivamente copiato **vColors** in **vCol**, vengono visualizzate tutte le **10** stringhe che formano lo stesso **vCol**.
La procedura **AreaCerchio** richiede un **Double** come argomento, e restituisce un altro **Double**; il valore restituito rappresenta l'area di un cerchio avente come raggio l'argomento passato a **AreaCerchio**.

Per vedere come lavorano le due procedure **InitScreen** e **InitRecCli**, esaminiamo il modulo **ASMMODA.ASM** che assume il seguente aspetto:

```
; file asmmoda.asm

; direttive per l'assembler
```

```
.386                                  ; set di istruzioni a 32 bit

; variabili definite in pasmod.pas

   EXTRN    strPas:            BYTE

; procedure definite in pasmod.pas

   EXTRN    WriteString:       FAR

; procedure definite nella unit Crt

   EXTRN    ClrScr:            FAR
   EXTRN    TextColor:         FAR
   EXTRN    TextBackground:    FAR

; segmento dati

DATA     SEGMENT  PARA PUBLIC USE16 'DATA'

DATA     ENDS

; segmento di codice

CODE     SEGMENT  PARA PUBLIC USE16 'CODE'

   PUBLIC   InitScreen
   PUBLIC   InitRecCli

   ASSUME   cs: CODE, ds: DATA

; procedure InitScreen(bgColor, fgColor: Byte);

InitScreen proc far

   fgColor  equ   [bp+6]
   bgColor  equ   [bp+8]

   push     bp
   mov      bp, sp

   ; selezione colore di sfondo

   push     word ptr bgColor
   call     far ptr TextBackground

   ; selezione colore di primo piano

   push     word ptr fgColor
   call     far ptr TextColor

   ; pulizia schermo

   call     ClrScr

   ; visualizzazione stringa

   push     seg strPas
   push     offset strPas
   push     word ptr 62
   call     far ptr WriteString
```

```
   pop      bp
   ret      4

InitScreen endp

; procedure InitRecCli(prc1, prc2: Pointer);

InitRecCli proc far

   prc2     equ   [bp+6]
   prc1     equ   [bp+10]

   push     bp
   mov      bp, sp
   push     ds

   ; ds:si = prc2 (sorgente)

   lds      si, dword ptr prc2

   ; es:di = prc1 (destinazione)

   les      di, dword ptr prc1

   ; copia prc2^ in prc1^

   mov      cx, 49
   cld
   rep      movsb

   pop      ds
   pop      bp
   ret      8

InitRecCli endp

CODE     ENDS

   END
```

E' necessario ribadire che e' fondamentale l'utilizzo del nome **CODE** o **_TEXT** per il blocco di codice contenente le procedure che devono essere visibili nei moduli **Pascal**; se non si utilizzano questi nomi, si ottiene un messaggio di errore da parte del compilatore.
Nel blocco delle direttive **EXTRN** osserviamo che **ASMMODA.ASM** e' in grado di vedere tutte le variabili e le procedure globali definite in **PASMOD.PAS**; inoltre, **ASMMODA.ASM** puo' vedere anche tutte le variabili e le procedure globali definite nelle varie **units** del **Turbo Pascal**. Notiamo infatti che **ASMMODA.ASM** utilizza tre procedure appartenenti alla **unit Crt**; a tale proposito il modulo **PASMOD.PAS** deve richiedere nella sezione **uses** l'inclusione di **Crt** nel programma.
Come abbiamo visto in precedenza, il primo compito svolto dal modulo **PASMOD.PAS** consiste nell'effettuare la chiamata:

```
InitScreen(Blue, Yellow);
```

Ricordando che il **Pascal** passa gli argomenti a partire dal primo e tenendo conto della **FAR** call, possiamo dire che il secondo parametro **fgColor** viene a trovarsi a **BP+6**, mentre il primo parametro **bgColor** viene a trovarsi a **BP+8**; questi due parametri sono di tipo **BYTE**, ma ovviamente vengono inseriti nello stack sotto forma di **WORD** con gli **8** bit piu' significativi che valgono **0**.
La procedura **InitScreen** svolge il proprio lavoro chiamando in successione le tre procedure **TextBackground**, **TextColor** e **ClrScr** definite nella **unit Crt**; per conoscere i prototipi di queste

procedure si puo' consultare l'help in linea del **Turbo Pascal**. Nel caso ad esempio di **TextBackground** abbiamo:

```
procedure TextBackground(Color: Byte);
```

L'unico argomento richiesto da **TextBackground** e' di tipo **Byte**; naturalmente, in un modulo **Assembly** e' compito del programmatore passare a **TextBackground** un argomento a **16** bit con gli **8** bit piu' significativi che valgono **0**. E' importante anche osservare che nel rispetto delle convenzioni **Pascal**, la pulizia dello stack viene delegata alla procedura chiamata; nel nostro caso, **TextBackground** provvedera' a togliere **2** byte dallo stack.
Prima di terminare **InitScreen** visualizza la stringa **strPas** attraverso la procedura **WriteString**; sia **strPas** che **WriteString** vengono definite nel modulo **PASMOD.PAS**. Il prototipo di **WriteString** e':

```
procedure WriteString(var s: String; d: Word);
```

Il parametro **s** e' l'indirizzo di una stringa, mentre il parametro **d** e' la distanza della fine della stringa dal bordo sinistro dello schermo; nel nostro caso la stringa viene visualizzata in modo che termini a **62** caratteri di distanza dal bordo sinistro dello schermo. Siccome stiamo lavorando con i puntatori **FAR**, la procedura **WriteString** deve ricevere l'indirizzo completo **seg:offset** di **strPas**; come al solito, e' importantissimo ricordare che la componente **seg** deve essere inserita per prima nello stack. Osserviamo anche che nel rispetto delle convenzioni **Pascal**, gli argomenti da passare a **WriteString** vengono inseriti nello stack a partire dal primo; infatti, prima viene inserito l'indirizzo **FAR** di **strPas**, e poi viene inserito il valore immediato **62** (a **16** bit).
La seconda procedura chiamata da **PASMOD.PAS** e' **InitRecCli** che richiede come argomenti due puntatori a **record** di tipo **recCliente**; questa procedura copia il **record** puntato da **prc2** (sorgente) nel **record** puntato da **prc1** (destinazione).
Anche in questo caso osserviamo subito che abbiamo a che fare con puntatori **FAR** contenenti ciscuno una coppia **seg:offset** da **16+16** bit; di conseguenza, il parametro **prc2** si viene a trovare a **BP+6**, mentre il parametro **prc1** si viene a trovare a **BP+10**. Per gestire questi due parametri vengono utilizzati i registri **DS:SI** come sorgente e **ES:DI** come destinazione; in questo modo possiamo utilizzare **REP MOVSB** per effettuare la copia ad altissima velocita'. E' importante ricordarsi anche di utilizzare l'istruzione **CLD** in modo da abilitare l'incremento automatico dei puntatori **SI** e **DI**; il contatore **CX** contiene il valore **49** che come abbiamo visto in precedenza e' la dimensione in byte dei **record** di tipo **recCliente**. Queste considerazioni ci fanno capire che per ottimizzare le prestazioni generali del programma, conviene assegnare alle stringhe dimensioni dispari; in questo modo, tenendo conto del **BYTE** di indice **0** aggiunto dal compilatore, si ottiene una stringa che occupa in memoria un numero pari di byte. Assegnando ad esempio la dimensione **41** al campo **nome** del **record** di tipo **recCliente**, si ottiene una dimensione complessiva pari a:
`2 + 4 + 42 = 48` byte di memoria.
Siccome **48** e' divisibile per **4** (**48 / 4 = 12**), all'interno della procedura **InitRecCli** possiamo caricare il valore **12** in **CX** e scrivere l'istruzione:

```
rep movsd
```

Questa istruzione come sappiamo e' mediamente **4** volte piu' veloce dell'analoga istruzione che utilizza **MOVSB**.
Per concludere l'analisi di **InitRecCli**, osserviamo che questa procedura modifica il registro **DS**, per cui e' importantissimo preservarne il contenuto originale; in caso contrario, al termine di **InitRecCli** il programma si pianta. La procedura **InitRecCli** ha ricevuto due argomenti da **4** byte ciascuno, e quindi deve terminare rimuovendo **8** byte dallo stack.

Le ultime due procedure chiamate dal modulo principale **PASMOD.PAS**, sono **UnitProcedure** e **AreaCerchio**; queste procedure vengono definite nel modulo **UNITMOD.PAS**. Per capire il funzionamento di **UnitProcedure** e di **AreaCerchio** analizziamo il contenuto della **unit UnitMod**.

```
{ file unitmod.pas }
```

```
{$F+}
{$L asmmodb.obj}

unit  UnitMod;

interface

{ dichiarazione tipi di dati }

type

   vStrType    = Array[0..9] of String[8];
   ptvStrType  = ^vStrType;

{ dichiarazione procedure pubbliche }

procedure UnitProcedure(vs1, vs2: ptvStrType);
function AreaCerchio(r: Double): Double;

implementation

{ dichiarazione tipi di dati }

type

   ptPrcType = procedure(var s: String);

{ variabili private }

const

   strUnit: String = 'Stringa definita nel modulo UNITMOD.PAS';

{ implementazione procedure pubbliche e private }

function AreaCerchio(r: Double): Double; external;
function StringLength(p: ptPrcType): Integer; external;
procedure InitvColors(vs1, vs2: ptvStrType); external;

procedure WriteString2(var s: String);
begin
   WriteLn(s);
end;

procedure UnitProcedure(vst1, vst2: ptvStrType);
var
   LungStr: Integer;
begin
   LungStr:= StringLength(WriteString2);
   WriteLn('Lunghezza stringa = ', LungStr);
   WriteLn;
   InitvColors(vst1, vst2);
end;

end.
```

Anche **UNITMOD.PAS** specifica la direttiva **{$F+}** che abilita le **FAR** calls; e' presente inoltre la direttiva **{$L}** che richiede il linking con **ASMMODB.OBJ** che contiene diverse procedure utilizzate da **UNITMOD.PAS**.
L'intestazione della **unit** e':
`unit UnitMod`

Come sappiamo, il nome della **unit** deve coincidere con il nome del file contenente la **unit** stessa. Nel nostro caso il compilatore produrra' una **unit** memorizzata in un file chiamato **UNITMOD.TPU**; in base all'intestazione che abbiamo utilizzato, tutto il codice di **UNITMOD.PAS** viene inserito in un segmento di programma avente le seguenti carattersitiche:

```
UnitMod SEGMENT PARA PUBLIC USE16 'CODE'
```

La sezione **interface** contiene le definizioni delle variabili globali pubbliche, e le dichiarazioni di tutte le procedure pubbliche fornite da questa **unit**; la sezione **implementation** contiene le definizioni delle variabili globali private e le definizioni di tutte le procedure pubbliche e private fornite da questa **unit**. Come gia' sappiamo, i moduli **Pascal** che usano **UnitMod** sono in grado di vedere solo gli identificatori presenti nella sezione **interface** di questa **unit**; i moduli **Assembly** facenti parte del programma possono invece vedere tutti gli identificatori globali, pubblici e privati presenti in **UnitMod**.
La prima procedura di **UNITMOD.PAS** chiamata da **PASMOD.PAS** e' **UnitProcedure**; il prototipo di questa procedura e' il seguente:

```
procedure UnitProcedure(vst1, vst2: ptvStrType);
```

Questa procedura richiede quindi due argomenti di tipo puntatore a **vStrType**; il contenuto dell'area di memoria puntata da **vst2** (sorgente), viene copiato nell'area di memoria puntata da **vst1** (destinazione). Per svolgere questo lavoro viene chiamata una apposita procedura **InitvColors** che viene definita nel modulo **ASMMODB.ASM**; prima di chiamare **InitvColors**, la procedura **UnitProcedure** mostra in pratica un esempio di chiamata ad una procedura che riceve come argomento un'altra procedura. A tale proposito viene chiamata la procedura **StringLength** definita sempre nel modulo **ASMMODB.ASM**; si tratta di una **function** che riceve come argomento l'indirizzo di una procedura di tipo **ptPrcType** e restituisce poi la lunghezza della stringa **strUnit** definita in **UNITMOD.PAS**.

Per analizzare il funzionamento di **StringLength** e di **InitvColors** dobbiamo fare riferimento al modulo **ASMMODB.ASM**; questo modulo presenta il seguente aspetto:

```
; file asmmodb.asm

; direttive per l'assembler

.386                                ; set di istruzioni a 32 bit

; variabili definite in unitmod.pas

   EXTRN    strUnit: BYTE

; segmento dati

DATA     SEGMENT  PARA PUBLIC USE16 'DATA'

DATA     ENDS

; segmento di codice

CODE     SEGMENT  PARA PUBLIC USE16 'CODE'

   PUBLIC   StringLength
   PUBLIC   InitvColors
   PUBLIC   AreaCerchio

   ASSUME   cs: CODE, ds: DATA

; function StringLength(ptp: ptPrcType): Integer;

StringLength proc far
```

```
   ptp      equ   [bp+6]

   push     bp
   mov      bp, sp

   ; visualizza strUnit

   push     seg strUnit
   push     offset strUnit
   call     dword ptr ptp

   ; ritorna la lunghezza di strUnit

   movzx    ax, byte ptr strUnit[0]

   pop      bp
   ret      4

StringLength endp

; procedure InitvColors(vs1, vs2: ptvStrType);

InitvColors proc far

   vs2      equ   [bp+6]
   vs1      equ   [bp+10]

   push     bp
   mov      bp, sp
   push     ds

   ; ds:si = vs2 (sorgente)

   lds      si, dword ptr vs2

   ; es:di = vs1 (destinazione)

   les      di, dword ptr vs1

   ; effettua la copia

   mov      cx, 90
   cld
   rep      movsb

   pop      ds
   pop      bp
   ret      8

InitvColors endp

; function AreaCerchio(r: Double): Double;

AreaCerchio proc far

   r        equ   [bp+6]

   push     bp
   mov      bp, sp

   finit                         ; inizializza la FPU
   fld      qword ptr r          ; ST(0) = r
```

```
   fst      st(1)                ; ST(1) = ST(0) = r
   fmul                          ; ST(0) = r * r
   fldpi                         ; ST(0) = pi_greco
   fmul                          ; ST(0) = pi_greco * (r * r)
   fwait                         ; sincronizzazione

   pop      bp
   ret      8

AreaCerchio endp

CODE     ENDS

   END
```

La procedura **StringLength** presenta il seguente prototipo:
```
function StringLength(ptp: ptPrcType): Integer;
```
Questa procedura riceve l'indirizzo **seg:offset** di un'altra procedura; in particolare, nel modulo **PASMOD.PAS** notiamo la chiamata:
```
StringLength(WriteString2);
```
All'interno di **StringLength** il parametro **ptp** rappresenta quindi l'indirizzo **seg:offset** di **WriteString2**; la chiamata di **WriteString2** e' di tipo indiretto intersegmento, per cui e' rappresentata dall'istruzione:
```
call dword ptr ptp
```
Come si puo' notare, **WriteString2** riceve come argomento la coppia **seg:offset** della stringa **strUnit** definita in **PASMOD.PAS**; a questo punto **WriteString2** provvede a visualizzare **strUnit** chiamando **WriteLn**. Il compito successivo svolto da **StringLength** consiste nel caricare in **AX** il suo valore di ritorno rappresentato dalla lunghezza di **strUnit**, e cioe' dal valore a **8** bit contenuto in **strUnit[0]**. Come si puo' notare, l'istruzione **MOVZX** azzera gli **8** bit piu' significativi di **AX**; prima di terminare **StringLength** toglie dallo stack **4** byte che rappresentano la dimensione della coppia **seg:offset** ricevuta come argomento.
La procedura **UnitProcedure** riceve in **AX** la lunghezza di **strUnit**; per dimostrarlo notiamo che **UnitProcedure** chiama **WriteLn** per visualizzare il valore di ritorno di **StringLength**.
La seconda procedura chiamata da **UnitProcedure** e' **InitvColors**; questa procedura copia un dato di tipo **vStrType** in un altro dato di tipo **vStrType**. Nel modulo **ASMMODB.ASM** possiamo notare che **InitvColors** e' del tutto simile alla procedura **InitRecCli** definita nel modulo **ASMMODA.ASM**; cio' dimostra che in **Assembly** possiamo gestire allo stesso modo un puntatore a **recCliente** e un puntatore a **vStrType** senza avere tra i piedi il controllo di tipo dei linguaggi di alto livello. E' chiaro infatti che in ogni caso, un puntatore **FAR** non e' altro che una coppia **seg:offset**; in pratica, potremmo riunire **InitvColors** e **InitRecCli** in un'unica procedura che riceve come terzo argomento la dimensione in byte dei blocchi da copiare. In relazione a **InitvColors** osserviamo anche che i dati di tipo **vStrType** occupano ciascuno **90** byte di memoria; invece di usare **MOVSB** con **CX=90**, possiamo allora utilizzare **MOVSW** con **CX=45**.
Il programma termina con la chiamata da parte di **PASMOD.PAS** della procedura **AreaCerchio** dichiarata in **UNITMOD.PAS** e definita in **ASMMODB.ASM**; si tratta di una **function** che riceve come argomento il raggio **r** di un cerchio, e restituisce come valore di ritorno l'area del cerchio di raggio **r**. Come si puo' notare in **ASMMODB.ASM**, questi calcoli vengono effettuati con la FPU; rispetto all'esempio presentato nel precedente capitolo, le uniche novita' sono rappresentate dall'uso delle due istruzioni **FST op** e **FLDPI**. L'istruzione **FST op** copia il contenuto del registro **ST(0)** nell'operando **op** che puo' essere anche un'altro registro della FPU; l'istruzione **FLDPI** carica in **ST(0)** il numero irrazionale **pi greco** (**3.14159267...**) in formato **Temporary Real** a **80** bit. Per il calcolo dell'area del cerchio viene utilizzata ovviamente la formula:
```
Area = pi_greco * (r * r)
```
Al termine del calcolo il risultato ottenuto e' a disposizione del caller nel registro **ST(0)** della **FPU**;

per dimostrarlo, nel modulo **PASMOD.PAS** viene visualizzato con **WriteLn** il valore di ritorno di **AreaCerchio**. Osserviamo infine che **AreaCerchio** ha ricevuto come argomento un **Double** che occupa **8** byte nello stack (**QWORD**); di conseguenza **AreaCerchio** deve terminare passando il valore immediato **8** all'istruzione **RET**.

Generazione dell'eseguibile

Come e' stato detto in precedenza, la fase di generazione dell'eseguibile e' condizionata dalle regole imposte dal compilatore **Turbo Pascal**; in generale, il procedimento da seguire si sviluppa nelle seguenti fasi:

***** Assemblaggio dei moduli **Assembly**
***** Compilazione delle **unit Pascal**
***** Compilazione del modulo principale **Pascal**
***** Generazione dell'eseguibile finale

E' importantissimo seguire quest'ordine per soddisfare tutte le dipendenze relative agli identificatori presenti nei vari moduli.
Per quanto riguarda l'assemblaggio dei moduli **Assembly** possiamo utilizzare sia **MASM** che **TASM**; e' fondamentale ricordarsi che gli **object files** prodotti dall'assembler devono essere in formato **OMF**. Con il **MASM32** non bisogna utilizzare quindi l'opzione **/coff**; inoltre l'assemblaggio deve essere **case-insensitive**, e quindi non bisogna utilizzare opzioni come **/ml** o **/Cp**.
La fase di compilazione dei moduli **Pascal** si puo' svolgere all'interno dell'**IDE** fornito dalla **Borland**; a tale proposito bisogna selezionare il menu **Compile**. Alternativamente e' anche possibile utilizzare il compilatore a linea di comando; questo strumento e' disponibile nella cartella **BIN** del **Turbo Pascal** ed e' rappresentato dal file **TPC.EXE**. Digitando **TPC** dalla linea di comando e premendo **[Invio]**, si ottiene l'elenco delle opzioni di configurazione del compilatore.

Nel caso del nostro esempio dobbiamo procedere in questo modo:

Se stiamo utilizzando **TASM**, con il comando:
```
tasm asmmoda.asm
```
generiamo il modulo **ASMMODA.OBJ**; con il comando:
```
tasm asmmodb.asm
```
generiamo il modulo **ASMMODB.OBJ**.
Se invece stiamo utilizzando **MASM**, con il comando:
```
ml /c asmmoda.asm
```
generiamo il modulo **ASMMODA.OBJ**; con il comando:
```
ml /c asmmodb.asm
```
generiamo il modulo **ASMMODB.OBJ**.

A questo punto, dall'interno dell'**IDE** procediamo alla configurazione del compilatore; attraverso il menu **Options - Compiler** dobbiamo abilitare l'uso delle istruzioni dell'**80286**, l'uso della FPU **80287** e l'uso delle **FAR** calls. Tutte queste opzioni possono essere specificate direttamente nei moduli **Pascal** grazie alle direttive **{$ }**; le precedenti tre opzioni equivalgono rispettivamente alle direttive **{$G+}**, **{$N+}** e **{$F+}**.
Terminata la fase di configurazione, attiviamo con il mouse la finestra contenente il modulo **UNITMOD.PAS** e selezioniamo il menu **Compile - Compile**; in questo modo otteniamo una **unit** memorizzata nel file **UNITMOD.TPU**.
Attiviamo ora con il mouse la finestra contenente il modulo **PASMOD.PAS** e selezioniamo il menu **Compile - Compile**; in questo modo otteniamo un modulo intermedio in formato **P-Code** che,

come e' stato detto in precedenza viene nascosto al programmatore. L'ultima fase consiste nel selezionare il menu **Compile - Make** che produce l'eseguibile finale chiamato **PASMOD.EXE**.

Allineamento dei dati al BYTE e alla WORD

Anche il **Turbo Pascal** permette di selezionare l'allineamento dei dati al **BYTE** o alla **WORD**; in questo modo e' possibile ottimizzare l'accesso ai dati da parte delle CPU con **Data Bus** formato da **16** o piu' linee. Per stabilire il tipo di allineamento per i dati e' necessario selezionare il menu **Options - Compiler**; nella finestra che compare bisogna attivare o disattivare la voce **Word align data**. In alternativa si puo' inserire direttamente nel programma la direttiva **{$A+}** che abilita l'allineamento dei dati alla **WORD**; viceversa, la direttiva **{$A-}** disabilita ogni forma di allineamento, in modo che i dati vengano disposti in memoria in modo consecutivo e contiguo. Consideriamo ad esempio il seguente blocco dati di un programma **Pascal**:

```
type
   vectorWord = array[0..7] of Word;

const
   i1:   Integer = 1200;
   b1:   Byte = 11;
   vw1:  vectorWord = (3, 6, 3, 9, 5, 3, 2, 1);
```

In presenza della direttiva **{$A-}**, la variabile **i1** a **16** bit viene disposta all'offset **0000h** del blocco dati, la variabile **b1** a **8** bit viene disposta all'offset **0002h** del blocco dati, mentre il vettore **vw1** viene fatto partire dall'offset **0003h** dello stesso blocco dati; in questo caso **vw1** parte da un offset dispari e la CPU ha bisogno di **2** accessi in memoria per leggere o scrivere ogni **WORD** del vettore.
In presenza invece della direttiva **{$A+}**, la variabile **i1** a **16** bit viene disposta all'offset **0000h** del blocco dati, la variabile **b1** a **8** bit viene disposta all'offset **0002h** del blocco dati, mentre il vettore **vw1** viene fatto partire dall'offset **0004h** dello stesso blocco dati; in questo caso **vw1** parte da un offset pari, e la CPU ha bisogno di **1** solo accesso in memoria per leggere o scrivere ogni **WORD** del vettore.
Nei moduli **Pascal** tutti questi aspetti vengono gestiti direttamente dal compilatore; nei moduli **Assembly** e' compito del programmatore tener conto dell'attributo di allineamento che e' stato selezionato per i dati **EXTRN** definiti in un modulo **Pascal**. In generale, e' vivamente sconsigliabile scrivere programmi **Assembly** che cercano di dedurre in modo empirico l'offset di una variabile; utilizzando invece l'istruzione **LEA** o l'operatore **OFFSET** si ottiene questa informazione in modo assolutamente sicuro ed affidabile.

30.7 Assembly inline

L'enorme successo ottenuto dal **Borland Turbo Pascal** e' dovuto anche alla eccezionale semplicita' ed efficienza che caratterizza il supporto dell'**Assembly inline** da parte di questo compilatore; il lavoro del programmatore viene anche agevolato dal fatto che il potente editor integrato del **Turbo**

Pascal permette di evidenziare con una apposita sintassi a colori i blocchi di istruzioni **Assembly inline** inseriti direttamente nel codice **Pascal**.
Le istruzioni **Assembly inline** del **Turbo Pascal** devono trovarsi all'interno di un blocco delimitato dalle due parole riservate **asm** e **end**; i blocchi di istruzioni **Assembly inline** possono comparire anche nel blocco di codice principale del programma. Come e' stato spiegato nel precedente capitolo, non si puo' pretendere che l'**Assembly inline** abbia la stessa potenza e flessibilita' dell'**Assembly** vero e proprio; in ogni caso, un programmatore **Assembly** esperto puo' fare quasi tutto anche con l'**Assembly inline**. Vediamo un semplice esempio pratico che mostra una procedura che copia una stringa sorgente in una stringa destinazione, e converte poi la stringa destinazione in maiuscolo; questo esempio e' contenuto nel seguente modulo **INLINE.PAS**:

```
{ file inline.pas }

Program AssemblyInline;

{ variabili globali inizializzate }

const

   vStr:    String = 'Stringa da convertire in maiuscolo';

{ variabili globali non inizializzate }

var

   vStr2:      String;
   Lunghezza:  Integer;

{ implementazione procedure }

function strToUpperCase(var d, s: String): Integer;
var
   Lung: Integer;
begin
   asm
      push     ds                     { preserva ds }
      lds      si, dword ptr s        { ds:si = sorgente }
      les      di, dword ptr d        { es:di = destinazione }
      xor      ch, ch                 { ch = 0 }
      mov      cl, ds:[si]            { cl = lunghezza s }
      mov      Lung, cx               { salva la lunghezza di s}
      inc      cx                     { cx = cx + 1 (per s[0]) }
      cld                             { clear direction flag }
      rep      movsb                  { copia s in d }
      pop      ds                     { ripristina ds }

      les      di, dword ptr d        { es:di = destinazione }
      xor      ch, ch                 { ch = 0 }
      mov      cl, es:[di]            { cl = lunghezza d }
      inc      cx                     { cx = cx + 1 (per d[0]) }
   @@toUpperLoop:
      mov      al, es:[di]            { char da esaminare }
      cmp      al, 'a'                { al < 'a' ? }
      jb       @@nextChar             { non e' una minuscola }
      cmp      al, 'z'                { al > 'z' ? }
      ja       @@nextChar             { non e' una minuscola }
      sub      byte ptr es:[di], 32 { converte in maiuscolo }
   @@nextChar:
      inc      di                     { prossimo char }
      loop     @@toUpperLoop          { controllo loop }
```

```
   end;
   strToUpperCase:= Lung;           { valore di ritorno }
end;

{ entry point del programma }

begin
   repeat
      Write('Inserisci una stringa: ');
      ReadLn(vStr);
      Lunghezza:= strToUpperCase(vStr2, vStr);
      Writeln('vStr  = ', vStr);
      WriteLn('vStr2 = ', vStr2);
      WriteLn('Lung  = ', Lunghezza);
   until (vStr = '');
end.
```

La procedura **strToUpperCase** richiede due argomenti che rappresentano gli indirizzi **seg:offset** di due dati di tipo **String**; il primo argomento e' la stringa destinazione, mentre il secondo argomento e' la stringa sorgente. Il lavoro svolto da questa procedura consiste nel copiare **s** (sorgente) in **d** (destinazione); successivamente la stringa destinazione viene convertita in maiuscolo attraverso l'algoritmo che gia' conosciamo. Il valore di ritorno di **strToUpperCase** e' un **Integer** che rappresenta la lunghezza della stringa appena convertita.
Come e' stato appena detto, siccome i due argomenti vengono passati per indirizzo, sia **d** che **s** rappresentano delle coppie **seg:offset**; l'indirizzo **s** (stringa sorgente) viene caricato in **DS:SI**, mentre l'indirizzo **d** (stringa destinazione) viene caricato in **ES:DI**. La lunghezza della stringa sorgente, e cioe' il contenuto del suo **BYTE** di indice **0**, viene caricato in **CX**; osserviamo che l'**Assembly inline** del **Turbo Pascal** non supporta le istruzioni a **32** bit, per cui non possiamo utilizzare ad esempio **MOVZX** per caricare direttamente un valore a **8** bit in **CX**. La lunghezza della stringa sorgente viene salvata nella variabile locale **Lung** che rappresenta anche il valore di ritorno della procedura; il contenuto di **CX** viene poi incrementato di **1** per tener conto anche del **BYTE** di indice **0** della stringa. A questo punto, la copia di **s** in **d** avviene con la solita istruzione **REP MOVSB**; nello svolgimento di questa fase e' importantissimo ricordarsi di preservare il contenuto di **DS**.
Nella fase di conversione di **d** in maiuscolo possiamo notare che il **Turbo Pascal** permette di inserire anche etichette locali all'interno di un blocco **asm end**; queste etichette devono essere precedute dal simbolo **@@** (doppia chiocciola). Osserviamo inoltre che come al solito, gli eventuali commenti devono seguire la sintassi del linguaggio di alto livello e non quella dell'**Assembly**.

Nel precedente capitolo abbiamo visto che in **C** un qualsiasi vettore viene sempre passato per indirizzo ad una procedura; il **Pascal** invece permette anche di passare un intero vettore per valore. Naturalmente si tratta di una scelta sconsigliabile in quanto, oltre ad influire negativamente sull'efficienza di un programma, comporta anche il rischio di saturazione dello stack; nel caso ad esempio del passaggio per valore di un vettore di **200 Integer**, vengono inserite nello stack ben **200 WORD** per un totale di **400** byte di dati.

Nel caso della procedura **strToUpperCase** puo' capitare che il programmatore voglia passare la stringa sorgente per valore in modo da evitare che il contenuto originale di questa stringa possa essere modificato dalla procedura stessa; in un caso del genere il prototipo di **strToUpperCase** diventa:

```
function strToUpperCase(var d: String; s: String): Integer;
```

Analizziamo ora il significato dei due parametri **d** e **s**; il parametro **d** rappresenta come al solito l'indirizzo completo **seg:offset** della stringa destinazione (in qualsiasi modello di memoria). La stringa sorgente invece e' stata passata per valore, per cui il compilatore ha creato nello stack una

copia di tutti i **256 BYTE** della stringa stessa; e' chiaro quindi che questa copia della stringa sorgente si trova nel segmento **STACK** referenziato da **SS**. L'offset iniziale di questa stringa e' rappresentato di conseguenza dall'offset del parametro **s** calcolato rispetto a **SS**; come gia' sappiamo, per ottenere questa informazione possiamo usare **LEA** scrivendo ad esempio:

```
lea si, s
```

A questo punto possiamo accedere alla copia della stringa sorgente tramite **SS:SI**; di conseguenza, i vari **BYTE** della stringa saranno rappresentati da **SS:[SI+0]**, **SS:[SI+1]**, **SS:[SI+2]** e cosi' via sino all'ultimo **BYTE** che e' **SS:[SI+255]**. In sostanza, se la stringa sorgente viene passata per valore, la copia di **s** in **d** puo' essere ottenuta in questo modo:

```
push     ds           { preserva ds }
push     ss           { copia ss }
pop      ds           { in ds }
lea      si, s        { ds:si punta a s }
les      di, d        { es:di punta a d }
xor      ch, ch       { ch = 0 }
mov      cl, ds:[si]  { cx = lunghezza s }
mov      Lung, cx     { salva la lunghezza di s }
inc      cx           { cx = cx + 1 }
cld                   { clear direction flag }
rep      movsb        { copia s in d }
pop      ds           { ripristina ds }
```

All'interno delle procedure che utilizzano l'**Assembly inline** si sconsiglia vivamente di utilizzare istruzioni del tipo:

```
lea si, [bp+6]
```

Infatti, quando si usa l'**Assembly inline**, lo **stack frame** delle procedure viene gestito direttamente dal compilatore che provvede anche a preservare il contenuto originale di **BP**; oltre a preservare **BP** il compilatore potrebbe anche inserire altre informazioni nello stack. Questo significa che non possiamo essere certi della esatta posizione nello stack dei parametri e della variabili locali di una procedura; utilizzando invece in modo esplicito i nomi simbolici associati ai parametri e alle variabili locali di una procedura, possiamo metterci al riparo da qualsiasi problema. Se al posto dell'istruzione precedente scriviamo ad esempio:

```
lea si, s
```

siamo sicuri che **LEA** carichera' in **SI** l'esatta posizione (offset) di **s** all'interno dello stack.
Le stesse considerazioni valgono ovviamente per i valori restituiti dalle **function**; nella fase di generazione dell'**epilog code** infatti, il compilatore potrebbe sovrascrivere i registri **AX** e **DX** che stiamo utilizzando per contenere un valore di ritorno. Per evitare problemi, le istruzioni per la restituzione del valore da parte di una **function** devono essere scritte in **Pascal** e non con l'**Assembly inline**; nel caso della procedura **strToUpperCase** possiamo notare che la lunghezza della stringa da convertire viene salvata nella variabile locale **Lung** in modo da poter scrivere alla fine:

```
strToUpperCase:= Lung;
```

In questo modo stiamo delegando al compilatore il compito di caricare **Lung** nel registro **AX**.
Un'ultima considerazione riguarda il fatto che anche con l'**Assembly inline** e' proibito qualsiasi riferimento agli identificatori definiti nella **unit System**; non e' possibile quindi scrivere istruzioni del tipo:

```
mov bx, offset WriteLn
```

Capitolo 32 - Interfaccia tra BASIC e Assembly

In questo capitolo vengono descritte le convenzioni che permettono di interfacciare il linguaggio **BASIC** con il linguaggio **Assembly**; si presuppone che chi legge abbia una adeguata conoscenza della programmazione in **BASIC**.

Il **BASIC** e' stato creato intorno al **1965** con lo scopo di insegnare la programmazione dei computers ai principianti assoluti, privi di qualsiasi conoscenza nel campo dell'informatica e dell'hardware; infatti **BASIC** e' l'acronimo di **Beginner's All purpose Symbolic Instruction Code** (linguaggio di programmazione simbolico e di uso generale per i principianti). Per raggiungere l'obiettivo di insegnare la programmazione a chiunque, il **BASIC** inizialmente venne privato di tutti quegli aspetti che avrebbero potuto mettere in difficolta' i principianti; in particolare, con le prime versioni del **BASIC** era possibile fare a meno di concetti come la tipizzazione dei dati e la programmazione strutturata. Purtroppo, con il passare degli anni queste caratteristiche si sono rivelate estremamente negative e controproducenti; la quasi totale mancanza di regole da rispettare, ha prodotto intere generazioni di programmatori di livello professionale molto basso. In particolare, la possibilita' di fare a meno della programmazione strutturata e l'uso sconsiderato dell'istruzione **GOTO**, ha favorito un modo di programmare contorto ed incomprensibile che e' stato subito etichettato con la tristemente famosa definizione di **spaghetti code**; nel tentativo di risolvere questi problemi, nel corso degli anni si e' assistito alla comparsa sul mercato di numerose nuove varianti del **BASIC** originario. Si e' passati cosi' dal **BASIC** vero e proprio ai vari **Turbo Basic** della **Borland**, al **Quick Basic** della **Microsoft**, etc; attualmente il **BASIC** sopravvive grazie al **Visual Basic** della **Microsoft** che permette di sviluppare applicazioni destinate all'ambiente **Windows**.
Un'altro aspetto legato alla evoluzione del **BASIC**, e' rappresentato dal fatto che le prime versioni di questo linguaggio di programmazione erano tutte di tipo interpretato; le versioni attuali (o quelle meno vecchie) del linguaggio **BASIC**, sono invece quasi tutte di tipo compilato. Come e' stato detto nei precedenti capitoli, l'interfacciamento con l'**Assembly** ha senso solo per i linguaggi compilati; per questo motivo, tutte le considerazioni esposte in questo capitolo si riferiscono al **Quick Basic 4.x** della **Miscrosoft** che nel bene e nel male rappresenta sicuramente la variante del **BASIC** che ha avuto maggiore successo in ambiente **DOS** a **16** bit.

31.1 Tipi di dati del BASIC

In precedenza e' stato detto che programmando in **BASIC** e' possibile fare a meno del concetto di tipizzazione dei dati; questo significa che in un qualunque punto di un programma **BASIC** possiamo scrivere:

```
PesoNetto = 1200
```

In questo modo abbiamo creato "al volo" una variabile apparentemente di tipo intero, che viene poi inizializzata con il valore **1200**; nel seguito del programma possiamo ora scrivere:

```
PesoNetto = PesoNetto + 4.5
```

In questo modo stiamo sommando un numero reale ad una variabile che dovrebbe essere di tipo intero; se proviamo a compilare questo programma, possiamo notare che non viene generato nessun messaggio di errore. A titolo di verifica possiamo usare l'istruzione:

```
PRINT "PesoNetto ="; PesoNetto
```

per constatare che effettivamente viene visualizzato sullo schermo il valore reale **1204.5**; questa situazione appare abbastanza strana in quanto non si capisce bene quale sia il tipo della variabile **PesoNetto** e quale sia la sua ampiezza in bit. Un programmatore **Assembly** pero' sa benissimo che la CPU ha bisogno di conoscere l'ampiezza in bit di un qualsiasi dato appartenente ad un programma scritto con un qualsiasi linguaggio di programmazione; per svelare questo mistero e' necessario allora analizzare il sistema che permette al **BASIC** di gestire in modo "occulto" i tipi dei dati di un programma.

La particolarita' del **BASIC** sta nel fatto che in questo linguaggio i tipi dei dati di un programma possono essere getiti in modo implicito; per capire questa situazione e' necessario premettere che il **BASIC** e' un linguaggio **case insensitive**, che non distingue quindi tra maiuscole e minuscole. Un identificatore del **BASIC**, cioe' un nome di una variabile, di una etichetta o di una procedura, deve iniziare obbligatoriamente con una tra le **26** possibili lettere dell'alfabeto inglese; al suo interno un nome puo' contenere esclusivamente lettere, cifre e il carattere '**.**' (punto). E' proibito invece l'uso di altri simboli come '**_**', '**@**', '**§**', etc; il programmatore puo' indicare esplicitamente il tipo di un identificatore di variabile o di funzione, posizionando alla fine del relativo nome uno tra i **5** appositi simboli: '**%**', '**&**', '**!**', '**#**' e '**$**'. In relazione al simbolo utilizzato si ha che:

***** un identificatore il cui nome termina con il simbolo '**%**' viene considerato esplicitamente di tipo **INTEGER** (intero con segno a **16** bit);
***** un identificatore il cui nome termina con il simbolo '**&**' viene considerato esplicitamente di tipo **LONG** (intero con segno a **32** bit);
***** un identificatore il cui nome termina con il simbolo '**!**' viene considerato esplicitamente di tipo **SINGLE** (reale **IEEE** a **32** bit in precisione singola);
***** un identificatore il cui nome termina con il simbolo '**#**' viene considerato esplicitamente di tipo **DOUBLE** (reale **IEEE** a **64** bit in precisione doppia);
***** un identificatore il cui nome termina con il simbolo '**$**' viene considerato esplicitamente di tipo **STRING** (vettore di codici ASCII da **8** bit ciascuno).

In assenza dei simboli che indicano esplicitamente il tipo di un identificatore di variabile o di funzione, il **BASIC** utilizza un metodo che permette di stabilire implicitamente una relazione tra la lettera iniziale del nome dell'identificatore e il relativo tipo; a tale proposito, ad ogni programma **BASIC** viene associata una tabella che assume il seguente aspetto:

```
! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Questa tabella indica il fatto che in assenza di diverse indicazioni da parte del programmatore, tutti gli identificatori privi di tipo, i cui nomi iniziano con una qualsiasi delle **26** lettere dell'alfabeto inglese, vengono considerati implicitamente di tipo **SINGLE** (reale a **32** bit); il **BASIC** in pratica aggiunge automaticamente il simbolo '**!**' a tutti i nomi degli identificatori privi di tipo.
Tutto cio' giustifica quello che e' successo con la variabile **PesoNetto**; il **BASIC** incontrando la definizione:
`PesoNetto = 1200`
la converte in:
`PesoNetto! = 1200.0` (variabile di tipo **SINGLE**).
Se vogliamo che **PesoNetto** sia esplicitamente di tipo **INTEGER** dobbiamo scrivere quindi:
`PesoNetto% = 1200`
Le considerazioni appena esposte valgono naturalmente anche per le funzioni che restituiscono un valore; nel caso ad esempio della dichiarazione:
`DECLARE FUNCTION Pitagora(lato1, lato2)`
tenendo conto della precedente tabella si ha che **Pitagora** e' una funzione che richiede due argomenti di tipo **SINGLE** e restituisce un valore di tipo **SINGLE**. Il **BASIC** infatti converte la precedente dichiarazione in:
`DECLARE FUNCTION Pitagora!(lato1!, lato2!)`

La tabella illustrata in precedenza ha lo scopo di permettere al programmatore di delegare al **BASIC** il compito di gestire implicitamente i tipi di dati; se il programmatore vuole alterare la situazione predefinita, puo' servirsi delle istruzioni **DEFINT**, **DEFLNG**, **DEFSNG**, **DEFDBL** e **DEFSTR**. Se in un programma **BASIC** scriviamo ad esempio:

```
DEFINT   A-G
DEFDBL   H-M
DEFSTR   N-T
```

allora la precedente tabella viene modificata nel modo seguente:

%	%	%	%	%	%	%	#	#	#	#	#	#	$	$	$	$	$	$	$	!	!	!	!	!	!
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

In questo caso:

***** un identificatore privo di tipo il cui nome inizia con le lettere comprese tra **A** e **G** e' implicitamente di tipo **INTEGER**;
***** un identificatore privo di tipo il cui nome inizia con le lettere comprese tra **H** e **M** e' implicitamente di tipo **DOUBLE**;
***** un identificatore privo di tipo il cui nome inizia con le lettere comprese tra **N** e **T** e' implicitamente di tipo **STRING**;
***** un identificatore privo di tipo il cui nome inizia con le lettere comprese tra **U** e **Z** e' implicitamente di tipo **SINGLE**;

Un programmatore proveniente dall'**Assembly** dal **C** o dal **Pascal**, leggendo queste cose non puo' che rabbrividire; questo sistema sembra concepito apposta per favorire gli errori di programmazione. Per rendercene conto, consideriamo il seguente programma **BASIC**:

```
DEFINT   A-Z

Raggio! = 25.37
PiGreco! = 3.14
AreaCerchio! = PiGreco * (Raggio * Raggio)
PRINT "AreaCerchio! ="; AreaCerchio!
END
```

Se si prova a compilare e ad eseguire questo programma, si puo' constatare che la **PRINT** visualizza il valore **0** al posto del previsto **2021.02**; quello che e' successo e' che il programmatore ha creato ed inizializzato due variabili di tipo esplicito **SINGLE**, chiamate **Raggio!** e **PiGreco!**. In seguito il programmatore ha creato una nuova variabile di tipo esplicito **SINGLE**, chiamata **AreaCerchio!**; per inizializzare questa nuova variabile il programmatore ha utilizzato **Raggio** e **PiGreco** dimenticando pero' il simbolo '**!**'. In base alla **DEFINT** inserita nel programma, il **BASIC** crea automaticamente due nuove variabili di tipo implicito **INTEGER**, chiamate appunto **Raggio** e **PiGreco**; in assenza di inizializzazione, tutte le variabili globali del **BASIC** vengono inizializzate con il valore **0** e questo giustifica il risultato visualizzato dalla **PRINT**. In sostanza, il programmatore a sua insaputa si ritrova con ben **5** variabili che sono: **Raggio!**, **PiGreco!**, **AreaCerchio!**, **Raggio%** e **PiGreco%**; per verificarlo possiamo aggiungere altre **PRINT** al programma precedente. La situazione e' ancora piu' assurda se si pensa che se commettiamo un errore di sintassi scrivendo ad esempio:

```
PRINT "Raggio% ="; Ragio%
```

allora il **BASIC** invece di segnalarci l'errore, crea una nuova variabile **Ragio%** di tipo esplicito

INTEGER e con valore iniziale **0**; tutto cio' ci fa capire per quale motivo, anche i programmatori **BASIC** piu' esperti si trovassero continuamente alle prese con stranissimi bugs che infestavano praticamente tutti i loro programmi.

Le considerazioni esposte in precedenza ci permettono di affermare che il **Quick Basic** fornisce i seguenti due tipi di dati interi:

Nome	Tipo	Val. Min.	Val. Max.
INTEGER	intero con segno a 16 bit	-32768	+32767
LONG	intero con segno a 32 bit	-2147483648	+2147483647

Come si puo' notare, non e' presente nessun tipo di dato intero a **8** bit e non vengono neanche supportati gli interi senza segno; viene fornito invece il supporto per i seguenti due tipi di dati in virgola mobile **IEEE**:

Nome	Tipo	Val. Min.	Val. Max.	Precisione
SINGLE	reale con segno a 32 bit	1.4×10^{-45}	3.4×10^{38}	7-8 cifre
DOUBLE	reale con segno a 64 bit	4.9×10^{-324}	1.7×10^{308}	15-16 cifre

Gli estremi **Min.** e **Max.** si riferiscono ai numeri reali positivi; per ottenere gli estremi negativi basta mettere il segno meno davanti agli estremi positivi.

Come e' stato detto in precedenza, il **BASIC** offre anche il supporto per le stringhe; una stringa **BASIC** e' come al solito un vettore di **BYTE** consecutivi e contigui. La dimensione di una stringa **BASIC** non puo' superare i **32767** byte di memoria; proprio il caso delle stringhe ci permette di analizzare una caratteristica molto particolare del **BASIC**. Supponiamo ad esempio di scrivere:

```
Stringa1$ = "Esempio di stringa BASIC"
```

Incontrando questa definizione il **BASIC** riserva nel segmento dati del programma, un blocco da **24** byte di memoria consecutivi e contigui, per contenere esattamente i **24** caratteri che formano **Stringa1$**; a questo punto bisogna capire come fa il **BASIC** a gestire la memoria riservata a questa stringa. Nei precedenti capitoli abbiamo visto ad esempio che il linguaggio **C** aggiunge ad ogni stringa uno **0** finale per delimitare la fine della stringa stessa; il **Pascal** invece utilizza il **BYTE** di indice **0** di una stringa per memorizzare la lunghezza della stringa stessa.
Il **BASIC** utilizza un sistema completamente differente che gli permette di gestire dinamicamente la memoria riservata alle stringhe e in generale ai dati di tipo vettoriale; per capire il funzionamento di questo sistema e' necessario premettere che in un programma **BASIC** esiste un unico blocco dati chiamato **DGROUP** e referenziato da **DS**. Ad ogni stringa presente nel blocco dati, il **BASIC** associa un cosiddetto **descrittore** di stringa; il **descrittore** di stringa viene creato ugualmente nel blocco dati ed e' costituito da due **WORD**. La **WORD** meno significativa contiene la lunghezza della stringa; la **WORD** piu' significativa contiene l'offset della stringa stessa, calcolato rispetto a **DS**. Supponendo ad esempio che **Stringa1$** venga posizionata in memoria all'indirizzo logico **DS:00B2h**, allora il suo **descrittore** conterra' la **DWORD 0018h:00B2h** (**18h = 24d**); come e' stato gia' detto, questo sistema permette al **BASIC** di gestire in modo dinamico la memoria riservata alle stringhe. Supponiamo ad esempio di modificare **Stringa1$** in questo modo:

```
Stringa1$ = "Esempio di stringa BASIC modificata"
```

In questo caso il **BASIC** libera i **24** byte riservati a **Stringa1$**, richiede nel blocco dati altri **35** byte consecutivi e contigui e li usa per contenere i **35** caratteri che formano la nuova **Stringa1$**; a questo punto il **BASIC** modifica il **descrittore** della stringa che diventa **0023h:00B2h** (**23h = 35d**).
Il vantaggio di questo sistema consiste nella possibilita' di sfruttare in modo ottimale lo spazio presente nell'unico blocco dati del programma; lo svantaggio e' rappresentato invece dal fatto che numerose allocazioni e deallocazioni della memoria possono portare alla frammentazione della

memoria stessa, proprio come accade nell'hard disk a causa delle frequenti operazioni di creazione e/o cancellazione di files. Per ovviare a questo inconveniente, il **memory manager** del **BASIC** supporta una nota caratteristica che prende il nome di **garbage collection** (raccolta della spazzatura); in sostanza, mentre il programma e' in esecuzione, il **BASIC** all'insaputa del programmatore puo' decidere ad un certo punto di effettuare un riordino (o deframmentazione) del blocco dati, che coinvolge principalmente i vettori monodimensionali (come le stringhe) e multidimensionali. In questo modo il blocco dati viene ricompattato con l'eliminazione nei limiti del possibile degli eventuali "buchi" presenti tra un dato e l'altro; naturalmente questa operazione coinvolge anche le eventuali stringhe, per cui il **BASIC** deve necessariamente aggiornare anche i relativi **descrittori**.
Dalle considerazioni appena esposte, emergono alcuni aspetti molto delicati che impongono al programmatore **Assembly** un comportamento ben preciso; in sostanza, e' necessario tenere sempre presente che a causa del **garbage collection**, l'indirizzo di determinati dati del **BASIC** non e' statico, ma puo' variare durante la fase di esecuzione del programma. Appare evidente quindi che utilizzando una procedura **Assembly** per manipolare in modo improprio i dati di un programma **BASIC**, possiamo danneggiare il sistema che il **BASIC** stesso utilizza per la gestione dei dati in memoria; nel seguito del capitolo vedremo come ci si deve comportare nei vari casi che si possono presentare.

31.2 Convenzioni per i compilatori BASIC

Segmenti di programma

In relazione al sistema di segmentazione da utilizzare nell'interfacciamento tra **BASIC** e **Assembly**, bisogna dire che la situazione viene notevolmente semplificata dal fatto che il compilatore **Quick Basic** e' in grado di generare gli **object files** in formato **OMF** (Object Module Format); tutto cio' ci permette di far interagire facilmente il **Quick Basic** con il **MASM** o con il **TASM**, per poter creare programmi estremamente flessibili. All'interno dei moduli **Assembly** possiamo inoltre creare segmenti di programma aventi attributi che possiamo scegliere liberamente in base alle nostre esigenze; solamente in rare circostanze siamo tenuti a creare segmenti di programma aventi attributi imposti dal **BASIC**. La conseguenza piu' importante di questa flessibilita' sta nel fatto che, come accade per il **C**, anche in questo caso abbiamo la possibilita' di condividere dati e procedure tra moduli **BASIC** e moduli **Assembly**; nel caso del **Turbo Pascal** abbiamo visto invece che l'impossibilita' di avere a disposizione gli **object files** dei moduli **Pascal** ci impone alcune limitazioni nell'interfacciamento con l'**Assembly**.

In generale, un programma interamente scritto in **BASIC** assume una struttura interna che equivale al modello di memoria **MEDIUM**; questo modello di memoria prevede come sappiamo la presenza di un unico blocco dati (comprendente anche lo stack), e di due o piu' segmenti di codice. In presenza di un unico blocco di dati, il **BASIC** gestisce le variabili globali del programma attraverso indirizzamenti di tipo **NEAR**; in presenza di due o piu' segmenti di codice, tutte le procedure di un programma **BASIC** devono essere di tipo **FAR**. Tutti i segmenti di dati appartenenti ai moduli **BASIC**, vengono inseriti dal compilatore in un gruppo chiamato **DGROUP** che in fase di esecuzione viene referenziato attraverso **DS**; in questo stesso gruppo viene anche inserito il segmento di stack del programma.
Nei moduli **Assembly** possiamo attenerci a queste regole creando uno o piu' segmenti di dati **NEAR** da inserire in **DGROUP**; come e' stato detto in precedenza, gli attributi di questi segmenti di dati possono essere scelti liberamente dal programmatore. Volendo utilizzare gli attributi standard che gia' conosciamo, possiamo creare ad esempio il seguente segmento di dati **NEAR**:
```
_DATA SEGMENT WORD PUBLIC USE16 'DATA'
```
In questo caso dobbiamo anche ricordarci di inserire la direttiva:

```
DGROUP GROUP _DATA
```
Nel blocco codice del modulo **Assembly** e' necessaria inoltre la direttiva:
```
ASSUME DS: DGROUP
```
Tutti i dati globali da condividere tra moduli **BASIC** e moduli **Assembly**, devono trovarsi in un segmento di dati **NEAR** avente i seguenti attributi:
```
COMMON SEGMENT PARA COMMON USE16 'BLANK'
```
In alternativa e' possibile scegliere un nome personalizzato; utilizzando ad esempio il nome **DATICOMUNI**, gli attributi di questo segmento diventano:
```
DATICOMUNI SEGMENT PARA COMMON USE16 'BC_VARS'
```
Anche questo blocco deve essere rigorosamente inserito in **DGROUP**; in presenza ad esempio del blocco **_DATA** e del blocco **COMMON** dobbiamo inserire la direttiva:
```
DGROUP GROUP _DATA, COMMON
```
Tutti i dettagli relativi al segmento dati condivisi verrano illustrati nel seguito del capitolo.
Eventualmente possiamo creare anche segmenti di dati **FAR** da non inserire in **DGROUP**; in questo caso il programma complessivo assume una struttura interna che equivale al modello di memoria **LARGE**. Utilizzando gli attributi standard possiamo creare ad esempio il seguente segmento di dati **FAR**:
```
FAR_DATA SEGMENT PARA PRIVATE USE16 'FAR_DATA'
```
Se vogliamo accedere ai dati **FAR** tramite **DS**, dobbiamo preservare rigorosamente il contenuto di questo registro di segmento; in base alle considerazioni appena esposte, e' importantissimo restituire sempre il controllo al **BASIC** con **DS=DGROUP**.

Come e' stato detto in precedenza, ciascun modulo **Assembly** collegato ad un programma **BASIC** puo' utilizzare un proprio segmento di codice; le procedure presenti in questi segmenti devono essere quindi tutte di di tipo **FAR**. Utilizzando gli attributi standard, in ogni modulo **Assembly** possiamo allora creare segmenti di codice del tipo:
```
name_TEXT SEGMENT WORD PUBLIC USE16 'CODE'
```
Avendo a disposizione le versioni piu' recenti di **MASM** e **TASM**, possiamo risparmiare parecchio lavoro attraverso le caratteristiche avanzate di questi due assembler; nel caso piu' generale possiamo inserire all'inizio di ogni modulo la direttiva:
```
.MODEL LARGE, BASIC
```
In questo modo possiamo creare i vari segmenti di programma attraverso le direttive semplificare **.DATA**, **.FARDATA** e **.CODE**; l'assembler provvedera' poi automaticamente a creare il gruppo **DGROUP** e ad inserire le necessarie direttive **ASSUME**. Osserviamo pero' che in presenza del blocco dati **COMMON**, tutti i relativi dettagli devono essere gestiti direttamente dal programmatore; il blocco **COMMON** infatti presenta caratteristiche non standard che non vengono quindi supportate da **MASM** e **TASM**.

Al momento di caricare il programma in memoria, il **SO** pone **SS=STACK=DGROUP**, mentre **CS** viene inizializzato con il segmento principale di codice **BASIC** che contiene ovviamente l'**entry point**; da parte sua il compilatore genera il codice macchina che pone automaticamente **DS=DGROUP**; a causa della presenza del gruppo **DGROUP**, tutte le procedure **Assembly** che modificano **DS** sono tenute a preservare rigorosamente il contenuto originale di questo registro di segmento.

Stack frame

Le convenzioni seguite dal **BASIC** per la gestione dello **stack frame** di una procedura, sono del tutto simili a quelle utilizzate dal **Pascal**; e' necessario inoltre ribadire che le chiamate alle procedure esterne sono tutte di tipo **FAR**.
Gli eventuali argomenti da passare ad una procedura vengono inseriti nello stack a partire dal primo (sinistra destra); all'interno dello stack gli argomenti di una procedura vengono quindi posizionati in ordine inverso rispetto a quello indicato dalla intestazione della procedura stessa. Il compito di

ripulire lo stack dagli argomenti spetta alla procedura chiamata; questo lavoro viene generalmente effettuato attraverso l'istruzione:

```
RET n
```

dove **n** indica il numero di byte da sommare a **SP**.
L'accesso ai parametri di una procedura avviene attraverso **BP**; e' necessario ricordare che i parametri sono posizionati in ordine inverso nello stack, e quindi muovendoci in avanti rispetto a **BP** incontreremo per primo l'ultimo parametro ricevuto dalla procedura. A causa delle **FAR** calls, l'ultimo parametro si trova sempre a **BP+6**.

Valori di ritorno

In relazione alle locazioni utilizzate per contenere gli eventuali valori di ritorno delle **FUNCTION** del **BASIC**, valgono le convenzioni illustrate nel **Capitolo 24**; anche nel **BASIC** quindi abbiamo la seguente situazione:

***** Gli interi a **8** bit vengono restituiti in **AL**.
***** Gli interi a **16** bit, compresi gli indirizzi **NEAR**, vengono restituiti in **AX**.
***** Gli interi a **32** bit vengono restituiti nella coppia **DX:AX**, con **DX** che in base alla notazione **little-endian** contiene la **WORD** piu' significativa.
***** Gli indirizzi **FAR** a **16+16** bit vengono restituiti sempre nella coppia **DX:AX**, con **DX** che contiene la componente **seg**, e **AX** la componente **offset**.

Per quanto riguarda la restituzione dei valori in virgola mobile **IEEE** di tipo **SINGLE** e **DOUBLE**, il **BASIC** utilizza invece le stesse convenzioni del **FORTRAN**; il procedimento che viene seguito consiste nei seguenti passi:

1) il caller crea nello stack un'area di memoria sufficiente a contenere il valore **IEEE** restituito dalla **FUNCTION**;
2) il caller chiama la **FUNCTION** passandole come ultimo argomento l'offset (che chiamiamo simbolicamente **OFFS**) dell'area di memoria appena creata nello stack (la componente **seg** e' ovviamente **SS**);
3) la **FUNCTION** a causa della **FAR** call trova **OFFS** a **BP+6** e gli eventuali altri parametri a partire da **BP+8**.
4) la **FUNCTION** memorizza in **SS:OFFS** il valore **IEEE** da restituire;
5) la **FUNCTION** prima di terminare carica **SS** in **DX** e **OFFS** in **AX**;
6) il caller riottiene il controllo e trova all'indirizzo **DX:AX** il valore **IEEE** restituito dalla **FUNCTION**.

31.3 Le classi di memoria del BASIC

Il **Quick Basic** fornisce una serie di caratteristiche che permettono di scrivere programmi adeguatamente strutturati; in particolare, vengono messi a disposizione diversi tipi di procedure che comprendono le **FUNCTION** e le **SUB**. La **FUNCTION** rappresenta la generalizzazione del concetto di operatore matematico e quindi fornisce sempre un valore di ritorno; la **SUB** rappresenta la generalizzazione del concetto di istruzione e non ha quindi nessun valore di ritorno.
Tutte le **FUNCTION** e le **SUB** definite in un modulo **BASIC** risultano automaticamente visibili anche nei moduli esterni; gli eventuali moduli **Assembly** possono quindi chiamare (con **FAR** calls) tutte le **FUNCTION** e le **SUB** presenti in un modulo **BASIC**. Tutte le procedure definite in un modulo **Assembly** e dichiarate **PUBLIC**, come gia' sappiamo risultano automaticamente visibili anche nei moduli esterni; queste procedure possono essere chiamate quindi (con **FAR** calls) sia da altri moduli **Assembly**, sia dal modulo principale **BASIC**.

Anche in relazione alle variabili di un programma, il **Quick Basic** ha introdotto una serie di innovazioni che permettono di creare variabili sia locali che globali; nelle prime versioni del **BASIC** invece tutte le variabili (anche quelle create all'interno di una procedura) erano visibili in qualsiasi punto del programma. Tutte le variabili globali definite in un modulo **BASIC** e non associate ad una direttiva **COMMON**, risultano visibili solo all'interno del modulo stesso; queste variabili globali possono essere paragonate alle variabili **static** del **C**. Tutte le variabili globali definite in un modulo **BASIC** e associate ad una direttiva **COMMON**, risultano invece visibili anche nei moduli esterni; queste variabili quindi possono essere condivise tra moduli **BASIC** e moduli **Assembly**.

Sempre in relazione alle variabili globali di un programma **BASIC**, analizziamo il metodo seguito dal compilatore per l'allocazione della relativa memoria; come e' stato gia' detto, un programma **BASIC** e' dotato di un unico blocco dati **DGROUP** che comprende: dati inizializzati, dati non inizializzati, dati costanti e stack. La dimensione del blocco **DGROUP** ovviamente non puo' essere superiore a **65536** byte; vediamo allora come sia possibile creare in un programma **BASIC** una quantita' di dati (in particolare grossi vettori multidimensionali), la cui dimensione complessiva puo' superare abbondantemente i **65536** byte.
Tutte le variabli numeriche semplici, e cioe' le variabili di tipo **INTEGER**, **LONG**, **SINGLE** e **DOUBLE**, vengono create nel blocco **DGROUP**; l'accesso a queste variabili avviene esclusivamente con indirizzamenti di tipo **NEAR** in quanto la loro componente **seg** e' implicitamente **DS=DGROUP**.
Tutte le variabili di tipo **STRING** vengono create nel blocco **DGROUP** e vengono associate come gia' sappiamo al relativo **descrittore** creato sempre in **DGROUP**; l'offset di una stringa contenuto nel relativo **descrittore** e' riferito quindi sempre a **DS=DGROUP**. Abbiamo anche visto che a causa del **garbage collection**, l'indirizzo delle stringhe del **BASIC** puo' variare dinamicamente in fase di esecuzione del programma; e' pericolosissimo quindi accedere alle stringhe **BASIC** in modo improprio. Se abbiamo ad esempio due stringhe **BASIC** chiamate **Stringa1$** e **Stringa2$**, e vogliamo copiare **Stringa2$** in **Stringa1$**, possiamo scrivere:

```
Stringa1$ = Stringa2$
```

Utilizzando invece metodi impropri, come ad esempio una procedura **Assembly** che accede direttamente a queste due stringhe, rischiamo di danneggiare il sistema di gestione della memoria del **BASIC**; in questi casi normalmente il programma viene interrotto e viene visualizzato il messaggio:

```
Spazio stringhe alterato
```

La memoria necessaria per i vettori del **BASIC** viene allocata in base al metodo che utilizziamo per specificare le dimensioni del vettore stesso; in questo modo e' possibile creare vettori gestiti staticamente o dinamicamente. Un vettore le cui dimensioni sono rappresentate da valori immediati, viene creato nel blocco **DGROUP**; questo caso si presenta ad esempio quando si scrive:

```
DIM Matr1%(5, 3)
```

In questo modo stiamo definendo un vettore **Matr1%** formato da **6** righe (indici da **0** a **5**) e **4** colonne (indici da **0** a **3**) per un totale di:

```
6 * 4 = 24
```

elementi di tipo **INTEGER** che richiedono complessivamente:

```
24 * 2 = 48
```

byte di memoria; questo blocco da **48** byte viene creato in **DGROUP**. L'accesso ad un vettore statico avviene con indirizzamenti di tipo **NEAR** in quanto la componente **seg** e' implicitamente **DS=DGROUP**; un vettore **BASIC** creato in questo modo non puo' essere ridimensionato con l'istruzione **REDIM**.
Supponiamo ora di aver definito le variabili **dim1% = 49** e **dim2% = 99**; a questo punto possiamo scrivere la definizione:

```
DIM Matr2!(dim1%, dim2%)
```

In questo modo stiamo definendo un vettore **Matr2!** formato da **50** righe (indici da **0** a **49**) e **100** colonne (indici da **0** a **99**), per un totale di:
```
50 * 100 = 5000
```
elementi di tipo **SINGLE** che richiedono complessivamente:
```
5000 * 4 = 20000
```
byte di memoria; questo blocco da **20000** byte viene creato nel **global heap** del programma al di fuori di **DGROUP**. Il **global heap** di un programma rappresenta tutta la memoria **convenzionale** disponibile; per conoscere questa informazione si puo' digitare il comando:
```
mem /c
```
dal prompt del **DOS**. L'accesso ad un vettore dinamico avviene con indirizzamenti di tipo **FAR** formati quindi da una coppia **seg:offset**; un vettore **BASIC** creato in questo modo puo' essere anche ridimensionato con l'istruzione **REDIM**.
Con il metodo appena descritto e' possibile persino creare vettori dinamici di tipo **HUGE** la cui dimensione complessiva puo' superare i **65536** byte; questi vettori vengono ovviamente creati nel **global heap** e vengono gestiti dal **BASIC** attraverso gli indirizzi logici normalizzati.

Anche i vettori del **BASIC** sono soggetti al **garbage collection**; proprio per questo motivo, in analogia con i dati di tipo **STRING**, il **BASIC** associa un **descrittore** ad ogni vettore che abbiamo definito nel programma. Come al solito, il **descrittore** viene creato nel blocco **DGROUP**; questo accade anche quando il vettore si trova nel **global heap**.

31.4 Gestione degli indirizzamenti in BASIC

Il **BASIC** e' un linguaggio di programmazione rivolto ai principianti assoluti, per cui tende a nascondere tutti i dettagli relativi agli indirizzamenti; contemporaneamente pero' il **BASIC** fornisce anche una serie di procedure di basso livello che permettono di operare direttamente con gli indirizzi di memoria.
Analizziamo innanzi tutto il sistema che il **BASIC** utilizza per il passaggio degli argomenti alle procedure; il comportamento predefinito del **BASIC** consiste nel passare gli argomenti sempre per indirizzo. Questo discorso vale quindi anche quando gli argomenti sono rappresentati da variabili semplici di tipo **INTEGER**, **LONG**, **SINGLE** e **DOUBLE**; vediamo un esempio pratico riferito alla dichiarazione:
```
DECLARE FUNCTION AddIntegers%(a%, b%)
```
Questa procedura richiede due argomenti di tipo **INTEGER** e termina restituendo in **AX** un **INTEGER** che rappresenta la somma tra **a%** e **b%**; per chiamare questa procedura utilizziamo le tre variabili globali **varint1% = 2000**, **varint2% = 5000** e **varint3%**. La chiamata di **AddIntegers%** e' rappresentata quindi dall'istruzione:
```
varint3% = AddIntegers%(varint1%, varint2%)
```
La procedura **AddIntegers%** riceve in questo caso, non i due valori **2000** e **5000**, ma le componenti **offset** di **varint1%** e **varint2%**; trattandosi di variabili semplici di tipo **INTEGER**, la loro componente **seg** e' implicitamente **DS=DGROUP**.

Se questa procedura e' scritta in **Assembly**, supponendo che **DS=DGROUP** avremo:

```
AddIntegers proc far

   b        equ   [bp+6]   ; offset di b
   a        equ   [bp+8]   ; offset di a

   push     bp             ; preserva bp
   mov      bp, sp         ; copia sp in bp

   mov      si, a          ; si = offset di a
   mov      di, b          ; di = offset di b

   mov      ax, [si]       ; ax = 2000
   add      ax, [di]       ; ax = 2000 + 5000

   pop      bp             ; ripristina bp
   ret      4              ; far return

AddIntegers endp
```

Come gia' sappiamo, il passaggio degli argomenti per indirizzo e' particolarmente veloce, soprattutto nel caso di argomenti di grosse dimensioni come le stringhe, i vettori, i record, etc; l'aspetto negativo e' rappresentato dal fatto che una procedura come **AddIntegers%** puo' inavvertitamente modificare il contenuto originale degli argomenti ricevuti. Se vogliamo servirci allora del passaggio per valore, dobbiamo indicarlo esplicitamente inserendo la parola chiave **BYVAL** nella dichiarazione della procedura; nel caso di **AddIntegers%** la dichiarazione diventa:

```
DECLARE FUNCTION AddIntegers%(BYVAL a%, BYVAL b%)
```

In questo modo **AddIntegers%** riceve i valori dei due argomenti e non gli indirizzi; la parola chiave **BYVAL** non puo' essere utilizzata nel caso di argomenti di tipo **STRING** o piu' in generale di tipo vettore. Nel caso di un argomento di tipo **STRING** il **BASIC** passa obbligatoriamente la componente **offset** del **descrittore** della stringa; anche in questo caso, la componente **seg** e' implicitamente **DS=DGROUP**.
Vediamo un esempio pratico che si riferisce ad una procedura **Assembly** che converte una stringa **BASIC** in maiuscolo; prima di tutto definiamo la seguente stringa:

```
Stringa1$ = "Stringa BASIC da convertire"
```

Supponiamo che questa stringa venga sistemata all'offset **3500** del blocco **DGROUP**; di conseguenza, il suo **descrittore** (creato anch'esso in **DGROUP**) sara' formato dalla **DWORD 27:3500**. Questa **DWORD** indica che **Stringa1$** e' lunga **27** caratteri e si trova in memoria all'indirizzo logico **DGROUP:3500**; passando **Stringa1$** come argomento di una procedura, cio' che viene passato e' l'offset del suo **descrittore** calcolato rispetto a **DGROUP**.
La dichiarazione **BASIC** della procedura del nostro esempio e' la seguente:

```
DECLARE SUB StrToUpper(s$)
```

La chiamata di **StrToUpper** con **Stringa1$** come argomento e' rappresentata dall'istruzione:

```
CALL StrToUpper(Stringa1$)
```

Sempre nell'ipotesi che **DS=DGROUP**, l'implementazione **Assembly** di questa procedura e' la seguente:

```
StrToUpper proc far

   s        equ   [bp+6]       ; offset descrittore

   push     bp                 ; preserva bp
   mov      bp, sp             ; copia sp in bp

   mov      bx, s              ; bx = offset descrittore
   mov      cx, [bx+0]         ; cx = lunghezza stringa
   mov      ax, [bx+2]         ; ax = offset inizio stringa
   mov      bx, ax             ; bx = ax

toUpperLoop:
   mov      al, [bx]           ; char da esaminare
   cmp      al, 'a'            ; confronto con 'a'
   jb       short nextChar     ; non e' una minuscola
   cmp      al, 'z'            ; confronto con 'z'
   ja       short nextChar     ; non e' una minuscola
   sub      byte ptr [bx], 32 ; converte in maiuscolo
nextChar:
   inc      bx                 ; prossimo char
   loop     toUpperLoop        ; controllo loop

   pop      bp                 ; ripristina bp
   ret      2                  ; far return

StrToUpper endp
```

Come e' stato gia' detto, questa procedura presuppone che **DS=DGROUP**; in questo caso, gli offset contenuti in **BX** vengono correttamente calcolati rispetto a **DGROUP**. Grazie al passaggio dell'argomento per indirizzo, la procedura **StrToUpper** puo' accedere direttamente al blocco di memoria contenente **Stringa1$** modificandone il contenuto originale; naturalmente, il programmatore deve evitare nella maniera piu' assoluta di alterare il contenuto del **descrittore** della stringa.

Analizziamo infine il caso degli argomenti di tipo vettore; anche in questo caso il **BASIC** passa obbligatoriamente l'indirizzo del vettore, e a differenza del **Pascal** non consente quindi il passaggio per valore di un intero vettore. In precedenza abbiamo visto che i vettori del **BASIC** possono essere creati staticamente in **DGROUP** e dinamicamente nel **global heap**; in questo secondo caso, la componente **seg** del vettore e' ovviamente diversa da **DGROUP**. Per gestire questa situazione, conviene scrivere procedure che ricevono come argomenti sia la componente **seg**, sia la componente **offset** del vettore; in questo modo possiamo operare indifferentemente su vettori statici e dinamici.
Per ottenere le componenti **seg** e **offset** di una qualunque variabile globale, il **BASIC** ci mette a disposizione le due funzioni **VARSEG** e **VARPTR**; la funzione **VARSEG** restituisce la componente **seg** (a **16** bit) del suo argomento, mentre la funzione **VARPTR** restituisce la componente **offset** (a **16** bit) del suo argomento.
Supponiamo ad esempio di aver definito una variabile globale **vardouble1#** che si trova all'offset **14** del blocco **DGROUP**, con **DS=DGROUP=3500**; in questo caso:

VARSEG(vardouble1#) restituisce il valore **3500**.
VARPTR(vardouble1#) restituisce il valore **14**.

Questo sistema funziona solo con il **Quick Basic 4.x**; nelle vecchie versioni del **Quick Basic** bisogna utilizzare invece la funzione **VARPTR** in combinazione con la procedura di libreria **PTR86**. Nella vecchia versione, la funzione **VARPTR** applicata ad una variabile, restituisce un valore a **32** bit che rappresenta la distanza (in byte) della variabile da **DS**; questo valore viene poi passato a **PTR86** che attraverso due altri argomenti passati per indirizzo, restituisce la coppia **seg:offset** della variabile. Nel caso ad esempio di **vardouble1#** dobbiamo scrivere:

```
CALL PTR86(seg%, offs%, VARPTR(vardouble1#))
```

Torniamo ora al caso del **Quick Basic 4.x** e vediamo come operano le due funzioni **VARSEG** e **VARPTR** nel caso di argomenti di tipo stringa e di tipo vettore; nel caso delle stringhe, queste due funzioni restituiscono le componenti **seg** e **offset** non della stringa ma, come al solito, del **descrittore** della stringa stessa.

Nel caso dei vettori, **VARSEG** e **VARPTR** restituiscono invece le componenti **seg** e **offset** dell'elemento specificato del vettore; nella maggior parte dei casi abbiamo bisogno di conoscere la coppia **seg:offset** del primo elemento di un vettore. Supponiamo allora di aver definito le due variabili **dim1% = 15** e **dim2% = 29**; a questo punto possiamo definire il seguente vettore bidimensionale di **INTEGER**:

```
DIM Matr1%(dim1%, dim2%)
```

Questo vettore e' formato da **16** righe e **30** colonne per un totale di **480** elementi; ogni elemento richiede due byte per cui **Matr1%** occupa in memoria **960** byte. In base alla precedente definizione, questi **960** byte verranno allocati dinamicamente nel **global heap** al di fuori quindi di **DGROUP**; supponiamo ora di voler scrivere una procedura **Assembly** che inizializza un generico vettore multimensionale di **INTEGER**, con un determinato valore iniziale. Prima di tutto inseriamo nel modulo **BASIC** una dichiarazione del tipo:

```
DECLARE SUB InitVector(BYVAL vseg%, BYVAL voff%, BYVAL n%, BYVAL i%)
```

I parametri **vseg%**, **voff%**, **n%** e **i%** rappresentano rispettivamente: la componente **seg** del vettore, la componente **offset**, il numero di elementi e il valore iniziale da assegnare ad ogni elemento; come si puo' notare, tutti gli argomenti vengono passati questa volta per valore. A questo punto, nel modulo **Assembly** possiamo scrivere la seguente procedura:

```
InitVector proc far

   i        equ   [bp+6]          ; valore iniziale
   n        equ   [bp+8]          ; numero elementi
   voff     equ   [bp+10]         ; componente offset
   vseg     equ   [bp+12]         ; componente seg

   push     bp                    ; preserva bp
   mov      bp, sp                ; copia sp in bp

   les      di, dword ptr voff    ; es:di = vseg:voff
   mov      ax, i                 ; ax = valore iniziale
   mov      cx, n                 ; cx = numero elementi
   cld                            ; DF = 0 (incremento)
   rep      stosw                 ; inizializza il vettore

   pop      bp                    ; ripristina bp
   ret      8                     ; far return

InitVector endp
```

Osserviamo che l'istruzione:

```
les di, dword ptr voff
```

trasferisce la coppia **vseg:voff** in **ES:DI**; questa istruzione lavora in modo corretto solo se **vseg** e' stata inserita nello stack prima di **voff**. Siccome il **BASIC** passa gli argomenti da sinistra a destra, nella lista dei parametri di **InitVector** e' necessario posizionare il parametro **vseg** prima di **voff**;

utilizzando invece le convenzioni del **C** avremmo dovuto invertire le posizioni di **vseg** e **voff**.
Se ora vogliamo inizializzare con il valore **10** i **480 INTEGER** di **Matr1%**, dobbiamo utilizzare la seguente chiamata **BASIC**:

```
CALL InitVector(VARSEG(Matr1%(0, 0)), VARPTR(Matr1%(0, 0)), 480, 10)
```

Come e' stato detto in precedenza, quando si utilizzano **VARSEG** e **VARPTR** con argomenti di tipo vettore, e' importantissimo indicare a quale elemento del vettore stesso ci si vuole riferire; nel nostro caso viene specificato naturalmente il primo elemento **Matr1%(0, 0)** del vettore.

Un'altro aspetto che assume una importanza fondamentale, e' rappresentato dal fatto che, come e' stato spiegato in precedenza, sia le stringhe, sia i vettori vengono associati ai rispettivi **descrittori**, e sono soggetti quindi al **garbage collection** da parte del **BASIC**; tutto cio' significa che in fase di esecuzione di un programma, l'area di memoria riservata a questi dati non e' fissa, ma puo' cambiare da un momento all'altro. Questa situazione impone al programmatore un comportamento ben preciso; in particolare, e' importante ricordare sempre che le informazioni restituite da funzioni come **VARSEG** e **VARPTR** devono essere utilizzate immediatamente. Queste informazioni infatti possono cambiare senza preavviso, e non avrebbe nessun senso quindi memorizzarle per poi utilizzarle in un secondo momento; in sostanza, **VARSEG** e **VARPTR** devono essere chiamate solo nel preciso momento in cui ci servono le informazioni restituite da queste due funzioni.
Un'altra conseguenza legata a questa delicata situazione, e' data dal fatto che un argomento di tipo stringa o vettore passato ad una procedura **BASIC** o **Assembly**, deve essere elaborato immediatamente; una procedura che riceve uno di questi argomenti e prima di elaborarlo lo passa ad un'altra procedura, sta seguendo un comportamento molto pericoloso. E' chiaro infatti che se la seconda procedura esegue ad esempio un'istruzione **REDIM** su un vettore ricevuto come argomento, restituisce alla prima procedura un vettore con differenti dimensioni e con differente indirizzo di memoria; se non si seguono queste precauzioni, si ottiene sicuramente un programma che funziona in modo anomalo.

Prima di concludere questa parte relativa agli indirizzamenti, e' necessario menzionare due istruzioni che i programmatori **BASIC** conoscono molto bene; si tratta naturalmente delle famigerate istruzioni **POKE** e **PEEK**, attraverso le quali e' possibile accedere in modo indiscriminato a qualsiasi area della memoria del computer, sia in lettura che in scrittura.
PEEK e' una funzione che richiede un argomento **offset** e restituisce un valore **byte**; l'argomento **offset** rappresenta un offset di memoria dal quale **PEEK** legge un valore a **8** bit. Il valore di ritorno **byte** rappresenta il valore a **8** bit letto da **PEEK**; possiamo dire quindi che **PEEK** ci fornisce il contenuto a **8** bit della locazione di memoria che si trova all'indirizzo **seg:offset**.
POKE e' un'istruzione che richiede due argomenti **offset** e **byte**; l'argomento **offset** rappresenta un offset di memoria nel quale **POKE** scrive un valore a **8** bit. L'argomento **byte** rappresenta il valore a **8** bit che **POKE** deve scrivere; possiamo dire quindi che **POKE** scrive il valore **byte** a **8** bit nella locazione di memoria che si trova all'indirizzo **seg:offset**.
A questo punto resta da capire rispetto a quale componente **seg** venga calcolata la componente **offset** utilizzata da **POKE** e **PEEK**; in assenza di altre indicazioni da parte del programmatore, la componente **offset** viene calcolata rispetto a **DGROUP**. Se il programmatore vuole alterare questa situazione, deve utilizzare l'istruzione **DEF SEG**; questa istruzione permette di stabilire la componente **seg** che verra' utilizzata dalle istruzioni **POKE**, **PEEK**, **BLOAD**, **BSAVE** e **CALL ABSOLUTE**.

Vediamo ad esempio come sia possibile inizializzare con il valore **10** (**&H000A**) tutti gli elementi del precedente vettore **Matr1%**; abbiamo visto che i **480 INTEGER** di **Matr1%** vengono allocati dinamicamente nel **global heap** al di fuori di **DGROUP**. Utilizzando **POKE** possiamo scrivere allora:

```
DEF SEG = VARSEG(Matr1%(0, 0))
MatrOffs% = VARPTR(Matr1%(0, 0))

k% = 0
FOR i% = 0 TO 479
   POKE (MatrOffs% + k%), &H0A      ' LSB
   POKE (MatrOffs% + k% + 1), &H00  ' MSB
   k% = k% + 2
NEXT i%

DEF SEG
```

Come si puo' notare, dopo aver utilizzato **POKE**, **PEEK**, **BLOAD**, etc, e' una buona regola ripristinare il segmento predefinito **DGROUP** per queste istruzioni; l'istruzione **DEF SEG** senza argomento ripristina appunto **DGROUP** come segmento predefinito per **POKE**, **PEEK**, **BLOAD**, etc.

31.5 Protocollo di comunicazione tra BASIC e Assembly

Analizziamo ora le regole che permettono a due o piu' moduli **BASIC** e **Assembly** di comunicare tra loro; l'insieme di queste regole definisce il protocollo di comunicazione tra **BASIC** e **Assembly**.

In relazione alle procedure di un programma, tutte le **FUNCTION** e le **SUB** definite in un modulo **BASIC** e associate alle rispettive **DECLARE**, risultano visibili anche nei moduli esterni come procedure di tipo **FAR**; un modulo **Assembly** che intende servirsi di queste procedure, le deve dichiarare **EXTRN** e le puo' cosi' chiamare attraverso **FAR** calls.
Tutte le procedure definite in un blocco codice di un modulo **Assembly** e dichiarate **PUBLIC**, risultano visibili anche nei moduli esterni; queste procedure per essere compatibili con le convenzioni del **BASIC** devono essere di tipo **FAR**. Un modulo **BASIC** che intende servirsi di queste procedure, le deve dichiarare con le apposite direttive **DECLARE** e le puo' cosi' chiamare attraverso **FAR** calls; naturalmente, in questo caso le **FAR** calls vengono gestite direttamente dal compilatore.

In relazione alle variabili di un programma, tutte le variabili globali definite in un modulo **BASIC** e non associate ad una direttiva **COMMON**, risultano invisibili nei moduli esterni; analogamente, tutte le variabili globali definite in un blocco dati di un modulo **Assembly** non compatibile con il blocco **COMMON** del **BASIC**, risultano invisibili nei moduli **BASIC** esterni.
Tutte le variabili globali definite o solo dichiarate in un modulo **BASIC** e associate ad una direttiva **COMMON**, risultano visibili anche nei moduli esterni; questa situazione si verifica ad esempio nel caso di un modulo **BASIC** che presenta la seguente dichiarazione:

```
COMMON num1%, num2%, num3!, num4!
```

In questo caso il **BASIC** crea il seguente segmento dati:

```
COMMON   SEGMENT  PARA COMMON USE16 'BLANK'

num1        dw    ?
num2        dw    ?
num3        dd    ?
num4        dd    ?

COMMON   ENDS
```

Un modulo **Assembly** che intende condividere queste variabili, deve definire un segmento dati avente gli stessi identici attributi del segmento appena illustrato; all'interno di questo segmento devono essere definite le stesse identiche variabili, nello stesso identico ordine specificato dalla direttiva **COMMON** del modulo **BASIC**. Questo blocco inoltre deve essere categoricamente inserito nel gruppo **DGROUP**; i dati in esso contenuti possono essere gestiti quindi con indirizzi di tipo **NEAR**.
Come si puo' notare, il precedente blocco **COMMON** presenta l'attributo di combinazione **COMMON**; nel Capitolo 12 abbiamo visto che tutti i segmenti di programma aventi lo stesso nome, la stessa classe e lo stesso attributo di combinazione **COMMON**, vengono sovrapposti tra loro in modo che condividano tutti lo stesso indirizzo fisico iniziale in memoria. Nel nostro caso, il blocco **COMMON** del **BASIC** e il blocco **COMMON** dell'**Assembly** vengono sovrapposti tra loro in modo da ottenere un unico blocco **COMMON** contenente le **4** variabili **num1%**, **num2%**, **num3!** e **num4!**; si tenga presente che in fase di compilazione, il **BASIC** rimuove dagli identificatori i simboli come **'%' '!'**, etc.
Il programmatore ha anche la possibilita' di assegnare un nome personalizzato al blocco **COMMON** appena descritto; nel modulo **BASIC** possiamo scrivere ad esempio:

```
COMMON /DATICOMUNI/ num1%, num2%, num3!, num4!
```

In questo caso il **BASIC** crea il seguente segmento dati:

```
DATICOMUNI  SEGMENT  PARA COMMON USE16 'BC_VARS'

num1        dw    ?
num2        dw    ?
num3        dd    ?
num4        dd    ?

DATICOMUNI  ENDS
```

Anche in questo caso, un modulo **Assembly** che intende condividere queste variabili, deve definire un segmento dati identico a quello appena illustrato; questo segmento deve poi essere inserito in **DGROUP**.

Tutte le procedure contenute nei moduli **Assembly** da linkare al **BASIC**, devono preservare rigorosamente il contenuto dei registri **CS**, **DS**, **SS**, **SP** e **BP**; tutti gli altri registri sono a completa disposizione del programmatore.

Il **BASIC** e' un linguaggio **case-insensitive**, e quindi i compilatori **BASIC** non distinguono tra maiuscole e minuscole; possiamo dire quindi che in **BASIC** i nomi come **varword1%**, **VarWord1%**, **VARWORD1%**, etc, rappresentano tutti lo stesso identificatore. A differenza pero' di quanto accade in **Pascal**, bisogna sempre ricordare che a causa della gestione implicita dei tipi dei dati, in un modulo **BASIC** la contemporanea presenza dei nomi **varword1%** e

VARWORD1%, non produce nessun messaggio di errore; il **BASIC** infatti considera i due nomi come identificatori della stessa variabile di tipo **INTEGER**.
In osservanza alle regole appena illustrate, al momento di assemblare un modulo **Assembly** da linkare ad un modulo **BASIC**, non dobbiamo assolutamente utilizzare le opzioni come **/ml** per il **TASM** o **/Cp** per il **MASM**.

31.6 Esempi pratici

Nella sezione **31.4** di questo capitolo abbiamo analizzato alcuni esempi che si riferiscono a procedure **Assembly** chiamabili da un modulo **BASIC**; vediamo ora un esempio completo che mostra anche come condividere variabili tra **BASIC** e **Assembly**, e come chiamare da un modulo **Assembly** una serie di **FUNCTION** e **SUB** scritte in **BASIC**.
Il programma di esempio e' formato dai due moduli **BASASM.BAS** e **ASMBAS.ASM**; il modulo **BASASM.BAS** ricopre il ruolo di modulo principale del programma, e presenta il seguente aspetto:

```
'----------------------------------------------'
' file basasm.bas                              '
' esempio di interfaccia tra Assembly e BASIC '
'----------------------------------------------'

DEFINT A-Z

' dichiarazione procedure esterne

DECLARE FUNCTION Pitagora! (sng1!, sng2!)
DECLARE SUB InitCommonVars ()
DECLARE SUB AsmProcedure (vint%, vsng!, vstr$)

' dichiarazione procedure interne

DECLARE FUNCTION Pitagora2! (sng1!, sng2!)
DECLARE SUB PrintInteger (vint%)
DECLARE SUB PrintSingle (vsng!)
DECLARE SUB PrintString (vstr$)

' dichiarazione variabili condivise

COMMON varint1%, varint2%, varsingle1!, varsingle2!

' definizione variabili globali

lato1! = 12.15232
lato2! = 27.39143

basint1% = 3500
bassng1! = 40.32167
basstr1$ = "Stringa definita nel modulo BASIC"

' inizializzazione variabili condivise

CALL InitCommonVars

' visualizzazione titolo del programma

CLS
PRINT "ESEMPIO DI INTERFACCIA TRA BASIC E ASSEMBLY"

PRINT
PRINT "Visualizzazione variabili condivise"
```

```
PRINT

PRINT "varint1%    = "; varint1%
PRINT "varint2%    = "; varint2%
PRINT "varsingle1! = "; varsingle1!
PRINT "varsingle2! = "; varsingle2!

PRINT
PRINT "SQR((lato1! * lato1!) + (lato2! * lato2!))"
PRINT

PRINT "Diagonale   = "; Pitagora!(lato1!, lato2!)

PRINT
PRINT "Visualizzazione variabili INTEGER, SINGLE e STRING"
PRINT

CALL AsmProcedure(basint1%, bassng1!, basstr1$)

END

FUNCTION Pitagora2! (sng1!, sng2!)
   Pitagora2! = SQR((sng1! * sng1!) + (sng2! * sng2!))
END FUNCTION

SUB PrintInteger (vint%)
   PRINT "INTEGER     = "; vint%
END SUB

SUB PrintSingle (vsng!)
   PRINT "SINGLE      = "; vsng!
END SUB

SUB PrintString (vstr$)
   PRINT "STRING      = "; vstr$
END SUB
```

Le varie **DECLARE** ci permettono di dichiarare sia le procedure esterne definite nel modulo **Assembly**, sia le procedure interne definite nel modulo **BASIC**; naturalmente, tutte le procedure esterne che vogliamo chiamare dal modulo **BASIC**, devono essere dichiarate **PUBLIC** nel modulo **Assembly** di appartenenza.
La direttiva **COMMON** presente nel modulo **BASASM.BAS** elenca una serie di variabili che vengono condivise con gli altri moduli del programma; e' importante ricordare che in un modulo **BASIC**, la direttiva **COMMON** deve precedere qualunque istruzione eseguibile, comprese le definizioni delle variabili globali.
Il primo compito svolto dal modulo **BASASM.BAS** consiste nel chiamare la **SUB InitCommonVars** per inizializzare le variabili condivise; questa **SUB** viene definita nel modulo **ASMMOD.ASM**. Subito dopo la loro inizializzazione, le variabili condivise vengono poi visualizzate attraverso una serie di **PRINT**.
Successivamente il modulo **BASASM.BAS** chiama la **FUNCTION Pitagora!** definita nel modulo **ASMBAS.ASM**; questa **FUNCTION** utilizza il teorema di Pitagora per calcolare la diagonale di un rettangolo avente per lati i due **SINGLE** passati come argomenti. In questo caso, la chiamata di **Pitagora!** viene gestita dal compilatore **BASIC** che quindi provvede anche a predisporre lo stack per ricevere il valore di tipo **SINGLE** restituito dalla **FUNCTION**; se invece la chiamata avviene da un modulo **Assembly**, questi dettagli sono a carico del programmatore.
L'ultimo compito svolto dal modulo **BASASM.BAS** consiste nella chiamata della **SUB AsmProcedure** definita nel modulo **ASMBAS.ASM**; questa **SUB** ha lo scopo di illustrare il metodo di chiamata di una serie di **FUNCTION** e **SUB** del **BASIC** che visualizzano valori di tipo

INTEGER, **SINGLE** e **STRING**.

Per chiarire meglio questi dettagli, analizziamo il seguente modulo **ASMBAS.ASM**:

```
;---------------------------------------------;
; file asmbas.asm                             ;
; esempio di interfaccia tra Assembly e BASIC ;
;---------------------------------------------;

;------------ direttive per l'assembler -------------

.386

; dichiarazione procedure definite nel modulo BASIC

   EXTRN    PrintInteger:  FAR
   EXTRN    PrintSingle:   FAR
   EXTRN    PrintString:   FAR
   EXTRN    Pitagora2:     FAR

;----------- blocco dati NEAR condivisi -------------

COMMON   SEGMENT  PARA COMMON USE16 'BLANK'

varint1     dw    ?
varint2     dw    ?
varsingle1  dd    ?
varsingle2  dd    ?

COMMON   ENDS

;------- blocco dati NEAR del modulo Assembly -------

_DATA    SEGMENT  WORD PUBLIC USE16 'DATA'

varfloat1   dd    35.14227
varfloat2   dd    46.35239

asmint1     dw    12539
asmsng1     dd    87.39261
asmstr1     dd    0
strAsm      db    'Stringa definita nel modulo ASM'
STRASMLEN   =     $ - offset strAsm

_DATA    ENDS

;------------------ gruppo DGROUP -------------------

DGROUP   GROUP    _DATA, COMMON

;------------------ blocco codice -------------------

_TEXT    SEGMENT  WORD PUBLIC USE16 'CODE'

   PUBLIC   Pitagora, InitCommonVars, AsmProcedure

   ASSUME   CS: _TEXT, DS: DGROUP

; FUNCTION Pitagora!(sng1!, sng2!)

Pitagora proc far
```

```
   retOffs  equ   [bp+6]            ; offset ret. val.
   sng2     equ   [bp+8]            ; offset sng2
   sng1     equ   [bp+10]           ; offset sng1

   push     bp                      ; preserva bp
   mov      bp, sp                  ; copia sp in bp

   mov      si, sng1                ; si = offset sng1
   mov      di, sng2                ; di = offset sng2
   mov      bx, retOffs             ; bx = offset ret. val.

   finit                            ; inizializza la FPU
   fld      dword ptr [si]          ; ST(0) = sng1
   fmul     dword ptr [si]          ; ST(0) = sng1 * sng1
   fld      dword ptr [di]          ; ST(0) = sng2
   fmul     dword ptr [di]          ; ST(0) = sng2 * sng2
   fadd                             ; ST(0) = ST(0) + ST(1)
   fsqrt                            ; ST(0) = SQRT(ST(0))
   fstp     dword ptr [bx]          ; ret. val. = ST(0)
   fwait                            ; sincronizza CPU e FPU

   mov      dx, ss                  ; dx = seg(ret. val.)
   mov      ax, bx                  ; ax = offset(ret. val.)

   pop      bp                      ; ripristina bp
   ret      6                       ; far return

Pitagora endp

; SUB InitCommonVars()

InitCommonVars proc far

   mov      varint1, 10             ; inizializza varint1
   mov      varint2, 15             ; inizializza varint2
   mov      eax, varfloat1          ; inizializza varsingle1
   mov      varsingle1, eax         ; con varfloat1
   mov      eax, varfloat2          ; inizializza varsingle2
   mov      varsingle2, eax         ; con varfloat2

   ret                              ; far return

InitCommonVars endp

; SUB AsmProcedure(vint%, vsng!, vstr$)

AsmProcedure proc far

   vstr     equ   [bp+6]            ; offset descritt. vstr
   vsng     equ   [bp+8]            ; offset vsng
   vint     equ   [bp+10]           ; offset vint
   retOffs  equ   [bp-4]            ; variabile locale

   push     bp                      ; preserva bp
   mov      bp, sp                  ; copia sp in bp
   sub      sp, 4                   ; spazio per retOffs

   push     vint                    ; passa l'offset di vint
   call     far ptr PrintInteger    ; visualizza vint

   push     offset dgroup:asmint1   ; passa l'offset di asmint1
   call     far ptr PrintInteger    ; visualizza asmint1
```

```
   push     vsng                    ; passa l'offset di vsng
   call     far ptr PrintSingle     ; visualizza vsng

   push     offset dgroup:asmsng1   ; passa l'offset di asmsng1
   call     far ptr PrintSingle     ; visualizza asmsng1

   push     offset dgroup:varfloat1 ; passa l'offset di varfloat1
   push     offset dgroup:varfloat2 ; passa l'offset di varfloat2
   lea      ax, retOffs             ; ax = offset retOffs
   push     ax                      ; passa l'offset di retOffs
   call     far ptr Pitagora2       ; chiama Pitagora2!
   push     ax                      ; passa ax = offset ret. value
   call     far ptr PrintSingle     ; visualizza [ax]

   push     vstr                    ; passa l'offset del descritt.
   call     far ptr PrintString     ; visualizza vstr

; simula un descrittore per strAsm e visualizza la stringa

   mov      word ptr asmstr1[0], STRASMLEN
   mov      word ptr asmstr1[2], offset dgroup:strAsm
   push     offset dgroup:asmstr1
   call     far ptr PrintString

   mov      sp, bp                  ; ripristina sp
   pop      bp                      ; ripristina bp
   ret      6                       ; far return

AsmProcedure endp

_TEXT       ENDS

;-----------------------------------------------------

   END
```

Il modulo **ASMBAS.ASM** inizia con una serie di dichiarazioni esterne per le procedure definite in **BASASM.BAS**; queste procedure esterne possono essere quindi chiamate da **ASMBAS.ASM** attraverso apposite **FAR** calls.
Subito dopo troviamo un blocco **COMMON** contenente le variabili **NEAR** condivise tra i vari moduli del programma; e' importante notare che nel blocco **COMMON** del modulo **ASMBAS.ASM**, queste variabili vengono definite con lo stesso nome, la stessa ampiezza in bit e la stessa disposizione specificata dalla direttiva **COMMON** del modulo **BASASM.BAS**.
Il modulo **ASMBAS.ASM** definisce anche un blocco riservato **_DATA** di variabili globali; questo blocco viene poi inserito insieme a **COMMON** in **DGROUP**, e puo' essere quindi gestito attraverso indirizzi di tipo **NEAR**.
Analizziamo ora il blocco codice **_TEXT** che contiene le definizioni di tutte le procedure chiamabili dal modulo **BASIC**; questo blocco contiene le necessarie direttive **PUBLIC** per queste procedure, e le direttive **ASSUME** che associano **CS** a **_TEXT** e **DS** a **DGROUP**.
La prima procedura chiamata da **BASASM.BAS** e' **InitCommonVars** che inizializza le variabili condivise; come si puo' notare, i due **SINGLE** condivisi **varsingle1** e **varsingle2** vengono inizializzati con i due **SINGLE** chiamati rispettivamente **varfloat1** e **varfloat2** definiti nel blocco **_DATA**.
La seconda procedura chiamata da **BASASM.BAS** e' **Pitagora!** che calcola la diagonale del rettangolo che ha per lati i due argomenti di tipo **SINGLE** ricevuti dalla procedura stessa; come gia' sappiamo, il comportamento predefinito del **BASIC** consiste nel passare l'offset degli argomenti e non il loro valore. Possiamo dire quindi che **Pitagora!** riceve gli offset **sng1** e **sng2** di due **SINGLE**; in entrambi i casi la componente **seg** e' implicitamente **DGROUP** in quanto i tipi numerici semplici del **BASIC** vengono sempre creati in questo gruppo.

Siccome **Pitagora!** restituisce un **SINGLE**, il **BASIC** prima di chiamare questa procedura predispone nello stack un'area da **4** byte per contenere il valore di ritorno; come gia' sappiamo, l'offset di quest'area viene passato come ultimo argomento a **Pitagora!**. Anche in questo caso, la componente **seg** predefinita e' **DGROUP**; infatti, in fase di inizializzazione del programma, il **BASIC** pone **DS=DGROUP** e **SS=DGROUP**. Tenendo conto delle convenzioni **BASIC** per il passaggio degli argomenti, troveremo quindi nello stack il parametro **retOffs** a **BP+6**, il parametro **sng2** a **BP+8** e il parametro **sng1** a **BP+10**; questi tre parametri rappresentano degli offset, per cui vengono gestiti dalla procedura attraverso i registri puntatori **BX**, **DI** e **SI**. Come al solito il calcolo della diagonale viene effettuato attraverso le istruzioni della **FPU**; nel rispetto delle convenzioni del **BASIC**, la procedura **Pitagora!** prima di terminare restituisce **SS** in **DX** e **retOffs** (cioe' **BX**) in **AX**.
Passiamo infine all'ultima procedura **AsmProcedure** chiamata da **BASASM.BAS**; questa procedura chiama a sua volta diverse altre procedure del modulo **BASIC** che svolgono svariati compiti, compresa la visualizzazione di dati di tipo **INTEGER**, **SINGLE** e **STRING**.
Prima di tutto **AsmProcedure** chiama **PrintInteger** per visualizzare dei dati di tipo **INTEGER**; come si nota dal prototipo, **PrintInteger** richiede l'offset di un dato di tipo **INTEGER**. Il primo **INTEGER** che viene visualizzato e' **vint** che e' stato ricevuto da **AsmProcedure** come parametro; questo parametro non e' altro che l'offset di un **INTEGER**, per cui puo' essere inserito direttamente nello stack e passato a **PrintInteger**. Per visualizzare invece **asmint1** definito nel blocco **_DATA**, dobbiamo calcolare necessariamente il suo offset; come al solito, per prevenire il bug dell'operatore **OFFSET**, dobbiamo inserire il segment override **dgroup:asmint1**.
La procedura **AsmProcedure** visualizza in seguito una serie di valori di tipo **SINGLE**; a tale proposito viene chiamata la procedura **PrintSingle** che in base al prototipo richiede l'offset di un **SINGLE**. Il primo **SINGLE** che viene visualizzato e' **vsng** che e' stato ricevuto da **AsmProcedure** come parametro; questo parametro non e' altro che l'offset di un **SINGLE**, per cui puo' essere inserito direttamente nello stack e passato a **PrintSingle**. Per visualizzare invece **asmsng1** definito nel blocco **_DATA**, dobbiamo calcolare necessariamente il suo offset; anche in questo caso, per prevenire il bug dell'operatore **OFFSET**, dobbiamo inserire il segment override **dgroup:asmsng1**.
AsmProcedure chiama **PrintSingle** anche per visualizzare un **SINGLE** restituito dalla procedura **Pitagora2!** definita nel modulo **BASASM.BAS** e identica alla procedura **Pitagora!** definita nel modulo **ASMBAS.ASM**; in questo modo abbiamo la possibilita' di analizzare la gestione dei valori di ritorno di tipo **IEEE** da parte di una procedura **Assembly**. Prima di tutto **AsmProcedure** crea nello stack un'area di **4** byte per contenere il **SINGLE** restituito da **Pitagora2!**; quest'area viene identificata dalla macro **retOffs**. La procedura **Pitagora2!** riceve quindi i tre argomenti rappresentati dall'offset di **varfloat1**, dall'offset di **varfloat2** e dall'offset di **retOffs**; naturalmente, per calcolare l'offset di **retOffs** utilizziamo l'istruzione **LEA**. La procedura **Pitagora2!** termina restituendo in **DX:AX** l'indirizzo **SS:retOffs** che contiene il valore di ritorno di tipo **SINGLE**; siccome **DX=SS** e **SS=DGROUP**, possiamo chiamare **PrintSingle** passandole **AX** che contiene l'offset del **SINGLE** da visualizzare.
L'ultimo compito svolto da **AsmProcedure** consiste nel chiamare **PrintString** per visualizzare delle stringhe **BASIC**; la procedura **PrintString** in base al suo prototipo richiede come sappiamo l'offset del **descrittore** della stringa da visualizzare. Il parametro **vstr** ricevuto da **AsmProcedure** e' gia' l'offset del **descrittore** di una stringa, per cui lo possiamo passare direttamente a **PrintString**; in seguito **AsmProcedure** chiama di nuovo **PrintString** per visualizzare una stringa **BASIC** "simulata" nel modulo **Assembly**. Come si puo' notare, nel blocco **_DATA** viene creato un "finto" **descrittore** chiamato **asmstr1** di tipo **DWORD**; in questo **descrittore** carichiamo la lunghezza della stringa **strAsm** e il relativo offset calcolato rispetto a **DGROUP**. A questo punto l'offset del **descrittore asmstr1** viene passato a **PrintString**; si consiglia in ogni caso di definire stringhe e vettori esclusivamente nei moduli scritti in **BASIC** in modo da garantire la corretta gestione del **garbage collection** da parte del **BASIC** stesso.

Un'ultima importantissima considerazione riguarda il fatto che tutti gli indirizzamenti relativi ai dati **NEAR** di un programma **BASIC**, si svolgono correttamente solo se **DS=DGROUP** e **SS=DGROUP**; nelle procedure **Assembly** e' importantissimo quindi preservare sempre il contenuto originale di questi registri di segmento.

Generazione dell'eseguibile

Come e' stato detto in precedenza, la generazione dell'eseguibile e' resa molto semplice dal fatto che il **Quick Basic** e' in grado di produrre gli **object files** dei moduli **BASIC**; il procedimento da seguire si sviluppa quindi nelle seguenti fasi:

***** Assemblaggio dei moduli **Assembly**
***** Compilazione del modulo principale **BASIC**
***** Generazione dell'eseguibile finale

Per quanto riguarda l'assemblaggio dei moduli **Assembly** possiamo utilizzare sia **MASM** che **TASM**; e' fondamentale ricordarsi che gli **object files** prodotti dall'assembler devono essere in formato **OMF**. Con il **MASM32** non bisogna utilizzare quindi l'opzione **/coff**; inoltre l'assemblaggio deve essere **case insensitive**, e quindi non bisogna utilizzare opzioni come **/ml** o **/Cp**.
Il compilatore a linea di comando del **Quick Basic** si chiama **BC.EXE**, mentre il linker a **16** bit si chiama **LINK.EXE** e puo' essere rimpiazzato da un qualunque altro linker a **16** bit della **Microsoft** (come quello fornito da **MASM16**). Non e' invece possibile utilizzare i linker a **16** bit della **Borland** o di altre marche; questo perche' gli **object files** prodotti dal **Quick Basic** contengono particolari caratteristiche interne che vengono supportate solo dai linker della **Microsoft**.
Vediamo ora come dobbiamo procedere per convertire **BASASM.BAS** e **ASMBAS.ASM** nell'eseguibile **BASASM.EXE**; supponiamo a tale proposito di aver installato il **TASM** nella cartella **C:\TASM**, e il **Quick Basic** nella cartella **C:\QB4**. Prima di tutto ci dobbiamo posizionare nella cartella **C:\QB4** dove si trovano i vari strumenti di sviluppo e le librerie del **BASIC**; a questo punto, supponendo che i moduli del programma si trovino ad esempio nella cartella di lavoro **C:\QB4\WORKBAS**, dobbiamo procedere in questo modo:

Con il comando:
```
c:\tasm\bin\tasm workbas\asmbas.asm
```
generiamo il modulo **ASMBAS.OBJ**.
Con il comando:
```
bc workbas\basasm.bas
```
generiamo il modulo **BASASM.OBJ**.
Con il comando:
```
link basasm.obj + asmbas.obj
```
generiamo l'eseguibile finale **BASASM.EXE**.

Per automatizzare tutto questo lavoro possiamo crearci un apposito **batch file**; a tale proposito, nella cartella **C:\QB4** creiamo ad esempio il seguente **batch file** chiamato **BASASM.BAT**:

```
echo off

rem GENERAZIONE DELL'ESEGUIBILE BASASM.EXE

rem assembling asmbas.asm

c:\tasm\bin\tasm workbas\asmbas.asm

rem compiling basasm.bas

bc workbas\basasm.bas

rem linking basasm.obj + asmbas.obj

link basasm.obj + asmbas.obj
```

Grazie a questo **batch file** possiamo eseguire le varie fasi digitando semplicemente **BASASM.BAT** dal **prompt** del **DOS**; anche in questo caso e' necessario posizionarsi nella cartella **C:\QB4**.
Se nel modulo **BASIC** vengono definiti vettori dinamici di tipo **HUGE**, e' necessario passare l'opzione **/ah** a **BC.EXE**; se vogliamo generare il **map file** del programma possiamo passare l'opzione **/map** a **LINK.EXE**.

Appendice A – Tabella dei codici ASCII estesi

American Standard Code for Information Interchange

Questo standard definisce un set di **256** simboli riconosciuto da tutti i **PC** in campo internazionale attraverso un codice a **8** bit. Questo significa che la stessa sequenza di codici **ASCII** verra' interpretata allo stesso modo da qualunque **PC** indipendentemente dalla nazionalita'; in particolare, i primi **128** simboli sono identici per tutti i **PC**, mentre i successivi **128** possono variare da nazione a nazione.
Le tabelle sottostanti contengono l'elenco dei **256** simboli e, per ciascuno di essi, il rispettivo codice **ASCII** in base **10**, **16** e **2**.

Nota: Per visualizzare correttamente questi simboli in ambiente **Windows**, bisogna disporre del font **Terminal**.

Codici di controllo			
Simbolo	**DEC**	**HEX**	**BIN**
NUL (Null)	000	00	00000000
SOH (Start of heading)	001	01	00000001
STX (Start of text)	002	02	00000010
ETX (End of text)	003	03	00000011
EOT (End of transmission)	004	04	00000100
ENQ (Enquiry)	005	05	00000101
ACK (Acknowledge)	006	06	00000110
BEL (Bell)	007	07	00000111
BS (Backspace)	008	08	00001000
HT (Horizontal tabulation)	009	09	00001001
LF (Line feed)	010	0A	00001010
VT (Vertical tabulation)	011	0B	00001011
FF (Form feed)	012	0C	00001100
CR (Carriage return)	013	0D	00001101
SO (Shift out)	014	0E	00001110
SI (Shift in)	015	0F	00001111
DLE (Data link escape)	016	10	00010000
DC1 (X-ON) (Device control one)	017	11	00010001
DC2 (TAPE) (Device control two)	018	12	00010010
DC3 (X-OFF) (Device control three)	019	13	00010011
DC4 (TAPE) (Device control four)	020	14	00010100
NAK (Negative acknowledge)	021	15	00010101
SYN (Synchronous idle)	022	16	00010110
ETB (End of transmission block)	023	17	00010111
CAN (Cancel)	024	18	00011000
EM (End of medium)	025	19	00011001
SUB (Substitute)	026	1A	00011010
ESC (Escape)	027	1B	00011011
FS (File separator)	028	1C	00011100

GS (Group separator)	029	1D	00011101
RS (Record separator)	030	1E	00011110
US (Unit separator)	031	1F	00011111

Simboli alfanumerici uguali per tutti i PC								
Simbolo	**DEC**	**HEX**	**BIN**		**Simbolo**	**DEC**	**HEX**	**BIN**
SPAZIO	032	20	00100000		P	080	50	01010000
!	033	21	00100001		Q	081	51	01010001
"	034	22	00100010		R	082	52	01010010
#	035	23	00100011		S	083	53	01010011
$	036	24	00100100		T	084	54	01010100
%	037	25	00100101		U	085	55	01010101
&	038	26	00100110		V	086	56	01010110
'	039	27	00100111		W	087	57	01010111
(	040	28	00101000		X	088	58	01011000
)	041	29	00101001		Y	089	59	01011001
*	042	2A	00101010		Z	090	5A	01011010
+	043	2B	00101011		[	091	5B	01011011
,	044	2C	00101100		\	092	5C	01011100
-	045	2D	00101101		]	093	5D	01011101
.	046	2E	00101110		^	094	5E	01011110
/	047	2F	00101111		_	095	5F	01011111
0	048	30	00110000		`	096	60	01100000
1	049	31	00110001		a	097	61	01100001
2	050	32	00110010		b	098	62	01100010
3	051	33	00110011		c	099	63	01100011
4	052	34	00110100		d	100	64	01100100
5	053	35	00110101		e	101	65	01100101
6	054	36	00110110		f	102	66	01100110
7	055	37	00110111		g	103	67	01100111
8	056	38	00111000		h	104	68	01101000
9	057	39	00111001		i	105	69	01101001
:	058	3A	00111010		j	106	6A	0110101
;	059	3B	00111011		k	107	6B	01101011
<	060	3C	00111100		l	108	6C	01101100
=	061	3D	00111101		m	109	6D	01101101
>	062	3E	00111110		n	110	6E	01101110
?	063	3F	00111111		o	111	6F	01101111
@	064	40	01000000		p	112	70	01110000
A	065	41	01000001		q	113	71	01110001
B	066	42	01000010		r	114	72	01110010
C	067	43	01000011		s	115	73	01110011
D	068	44	01000100		t	116	74	01110100
E	069	45	01000101		u	117	75	01110101

F	070	46	01000110		v	118	76	01110110
G	071	47	01000111		w	119	77	01110111
H	072	48	01001000		x	120	78	01111000
I	073	49	01001001		y	121	79	01111001
J	074	4A	01001010		z	122	7A	01111010
K	075	4B	01001011		{	123	7B	01111011
L	076	4C	01001100		|	124	7C	01111100
M	077	4D	01001101		}	125	7D	01111101
N	078	4E	01001110		~	126	7E	01111110
O	079	4F	01001111		DEL	127	7F	01111111

Simboli alfanumerici dipendenti dalla nazionalita'								
Simbolo	**DEC**	**HEX**	**BIN**		**Simbolo**	**DEC**	**HEX**	**BIN**
Ç	128	80	10000000		+	192	C0	11000000
ü	129	81	10000001		-	193	C1	11000001
é	130	82	10000010		-	194	C2	11000010
â	131	83	10000011		+	195	C3	11000011
ä	132	84	10000100		-	196	C4	11000100
à	133	85	10000101		+	197	C5	11000101
å	134	86	10000110		ã	198	C6	11000110
ç	135	87	10000111		Ã	199	C7	11000111
ê	136	88	10001000		+	200	C8	11001000
ë	137	89	10001001		+	201	C9	11001001
è	138	8A	10001010		-	202	CA	11001010
ï	139	8B	10001011		-	203	CB	11001011
î	140	8C	10001100		¦	204	CC	11001100
ì	141	8D	10001101		-	205	CD	11001101
Ä	142	8E	10001110		+	206	CE	11001110
Å	143	8F	10001111		¤	207	CF	11001111
É	144	90	10010000		ð	208	D0	11010000
æ	145	91	10010001		Ð	209	D1	11010001
Æ	146	92	10010010		Ê	210	D2	11010010
ô	147	93	10010011		Ë	211	D3	11010011
ö	148	94	10010100		È	212	D4	11010100
ò	149	95	10010101		i	213	D5	11010101
û	150	96	10010110		Í	214	D6	11010110
ù	151	97	10010111		Î	215	D7	11010111
ÿ	152	98	10011000		Ï	216	D8	11011000
Ö	153	99	10011001		+	217	D9	11011001
Ü	154	9A	10011010		+	218	DA	11011010
ø	155	9B	10011011		_	219	DB	11011011
£	156	9C	10011100		_	220	DC	11011100
Ø	157	9D	10011101		¦	221	DD	11011101
×	158	9E	10011110		Ì	222	DE	11011110

ƒ	159	9F	10011111		_	223	DF	11011111
á	160	A0	10100000		Ó	224	E0	11100000
í	161	A1	10100001		ß	225	E1	11100001
ó	162	A2	10100010		Ô	226	E2	11100010
ú	163	A3	10100011		Ò	227	E3	11100011
ñ	164	A4	10100100		õ	228	E4	11100100
Ñ	165	A5	10100101		Õ	229	E5	11100101
ª	166	A6	10100110		µ	230	E6	11100110
º	167	A7	10100111		þ	231	E7	11100111
¿	168	A8	10101000		Þ	232	E8	11101000
®	169	A9	10101001		Ú	233	E9	11101001
¬	170	AA	10101010		Û	234	EA	11101010
½	171	AB	10101011		Ù	235	EB	11101011
¼	172	AC	10101100		ý	236	EC	11101100
¡	173	AD	10101101		Ý	237	ED	11101101
«	174	AE	10101110		¯	238	EE	11101110
»	175	AF	10101111		´	239	EF	11101111
_	176	B0	10110000			240	F0	11110000
_	177	B1	10110001		±	241	F1	11110001
_	178	B2	10110010		_	242	F2	11110010
¦	179	B3	10110011		¾	243	F3	11110011
¦	180	B4	10110100		¶	244	F4	11110100
Á	181	B5	10110101		§	245	F5	11110101
Â	182	B6	10110110		÷	246	F6	11110110
À	183	B7	10110111		¸	247	F7	11110111
©	184	B8	10111000		°	248	F8	11111000
¦	185	B9	10111001		¨	249	F9	11111001
¦	186	BA	10111010		·	250	FA	11111010
+	187	BB	10111011		¹	251	FB	11111011
+	188	BC	10111100		³	252	FC	11111100
¢	189	BD	10111101		²	253	FD	11111101
¥	190	BE	10111110		_	254	FE	11111110
+	191	BF	10111111		SPAZIO	255	FF	11111111

Appendice B – Set di istruzioni della CPU

Note:

Questo documento rappresenta un manuale di riferimento rapido al set di istruzioni della CPU; per ogni istruzione viene fornito:

a) l'opcode principale;
b) i cicli di clock per le CPU 8086, 80286, 80386, 80486;
c) lo stato dei flags dopo l'esecuzione dell'istruzione.

Per i flags si utilizzano le seguenti abbreviazioni:

O = Overflow Flag
D = Direction Flag
I = Interrupt Enable Flag
T = Trap Flag
S = Sign Flag
Z = Zero Flag
A = Auxiliary Flag
P = Parity Flag
C = Carry Flag

Lo stato dei flags dopo l'esecuzione di una istruzione, viene descritto con i seguenti simboli:
- = Il flag non viene modificato.
? = Il flag e' indefinito.
x = Il flag cambia in base al risultato prodotto dall'istruzione.
0 = Il flag viene posto a zero.
1 = Il flag viene posto a uno.

Abbreviazioni utilizzate per gli operandi delle istruzioni:

SRC = operando sorgente (source)
DEST = operando destinazione (destination)
r8, r16, r32 = registro a 8 / 16 / 32 bit
m8, m16, m32 = blocco di memoria da 8 / 16 / 32 bit
i8, i16, i32 = valore immediato a 8 / 16 / 32 bit
o8, o16, o32 = indirizzo di memoria (offset) a 8 / 16 /32 bit
p16:16, p16:32 = indirizzo di memoria in formato seg:offset
sr = segment register (registro di segmento a 16 bit)
rel8, rel16, rel32 = valore relativo a 8 / 16 /32 bit

Nel calcolo dei cicli di clock necessari alla CPU (in modalita' reale) per l'esecuzione di ogni istruzione, si considerano valide le seguenti condizioni:
- l'istruzione e' gia' stata precaricata e decodificata ed, e' quindi pronta per l'esecuzione;
- non sono necessari stati di attesa (wait states) della CPU;
- gli eventuali operandi che si trovano in memoria, sono correttamente allineati.

Le CPU prese in considerazione sono quelle considerate standard di riferimento:
- 8086 a 5 MHz;
- 80286 a 10 MHz;
- 80386 DX a 20 MHz;
- 80486 DX a 33 MHz.

Per conoscere i cicli di clock relativi alle CPU di classe superiore (80568, 80686, etc), si faccia riferimento ai relativi manuali tecnici forniti dai produttori.

A

AAA

ASCII adjust AL after addition – Aggiustamento del contenuto di **AL** dopo una addizione tra due cifre **Unpacked BCD**.

Flags									**Opcode**	**Istruzione**	**Clocks**			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
?	-	-	-	?	?	x	?	x	37h	AAA	8	3	4	3

AAD

ASCII adjust AX before division – Aggiustamento del contenuto di **AX** prima di una divisione tra numeri **Unpacked BCD**.

Flags									**Opcode**	**Istruzione**	**Clocks**			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
?	-	-	-	x	x	?	x	?	D50Ah	AAD	60	14	19	14

AAM

ASCII adjust AX after multiplication – Aggiustamento del contenuto di **AX** dopo una moltiplicazione tra due cifre **Unpacked BCD**.

Flags									**Opcode**	**Istruzione**	**Clocks**			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
?	-	-	-	x	x	?	x	?	D40Ah	AAM	83	16	17	15

AAS

ASCII adjust AL after subtraction – Aggiustamento del contenuto di **AL** dopo una sottrazione tra due cifre **Unpacked BCD**.

Flags									**Opcode**	**Istruzione**	**Clocks**			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
?	-	-	-	?	?	x	?	x	3Fh	AAS	8	3	4	3

ADC

Add with carry - Esegue la somma:
DEST = DEST + SRC + CF.

Flags								
O	D	I	T	S	Z	A	P	C
x	-	-	-	x	x	x	x	x

Opcode	Istruzione	Clocks			
		8086	80286	80386	80486
14h	ADC AL,i8	4	3	2	1
15h	ADC AX,i16	4	3	2	1
15h	ADC EAX,i32	-	-	2	1
80h	ADC r8/m8,i8	4/17+EA	3/7	2/7	1/3
81h	ADC r16/m16,i16	4/17+EA	3/7	2/7	1/3
81h	ADC r32/m32,i32	-	-	2/7	1/3
83h	ADC r16/m16,i8	4/17+EA	3/7	2/7	1/3
83h	ADC r32/m32,i8	-	-	2/7	1/3
10h	ADC r8/m8,r8	3/16+EA	2/7	2/7	1/3
11h	ADC r16/m16,r16	3/16+EA	2/7	2/7	1/3
11h	ADC r32/m32,r32	-	-	2/7	1/3
12h	ADC r8,r8/m8	3/9+EA	2/7	2/6	1/2
13h	ADC r16,r16/m16	3/9+EA	2/7	2/6	1/2
13h	ADC r32,r32/m32	-	-	2/6	1/2

ADD

Addition - Esegue la somma:
DEST = DEST + SRC.

Flags								
O	D	I	T	S	Z	A	P	C
x	-	-	-	x	x	x	x	x

Opcode	Istruzione	Clocks			
		8086	80286	80386	80486
04h	ADD AL,i8	4	3	2	1
05h	ADD AX,i16	4	3	2	1
05h	ADD EAX,i32	-	-	2	1
80h	ADD r8/m8,i8	4/17+EA	3/7	2/7	1/3
81h	ADD r16/m16,i16	4/17+EA	3/7	2/7	1/3
81h	ADD r32/m32,i32	-	-	2/7	1/3
83h	ADD r16/m16,i8	4/17+EA	3/7	2/7	1/3
83h	ADD r32/m32,i8	-	-	2/7	1/3
00h	ADD r8/m8,r8	3/16+EA	2/7	2/7	1/3
01h	ADD r16/m16,r16	3/16+EA	2/7	2/7	1/3
01h	ADD r32/m32,m32	-	-	2/7	1/3
02h	ADD r8,r8/m8	3/9+EA	2/7	2/6	1/2
03h	ADD r16,r16/m16	3/9+EA	2/7	2/6	1/2
03h	ADD r32,r32/m32	-	-	2/6	1/2

AND

Logical AND - Esegue un **AND** logico tra **SRC** e **DEST**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
0	-	-	-	x	x	?	x	0	24h	AND AL,i8	4	3	2	1
									25h	AND AX,i16	4	3	2	1
									25h	AND EAX,i32	-	-	2	1
									80h	AND r8/m8,i8	4/17+EA	3/7	2/7	1/3
									81h	AND r16/m16,i16	4/17+EA	3/7	2/7	1/3
									81h	AND r32/m32,i32	-	-	2/7	1/3
									83h	AND r16/m16,i8	4/17+EA	3/7	2/7	1/3
									83h	AND r32/m32,i8	-	-	2/7	1/3
									20h	AND r8/m8,r8	3/16+EA	2/7	2/7	1/3
									21h	AND r16/m16,r16	3/16+EA	2/7	2/7	1/3
									21h	AND r32/m32,r32	-	-	2/7	1/3
									22h	AND r8,r8/m8	3/9+EA	2/7	2/6	1/2
									23h	AND r16,r16/m16	3/9+EA	2/7	2/6	1/2
									23h	AND r32,r32/m32	-	-	2/6	1/2

B

BOUND

Check array index against bounds - Verifica dei limiti dell'indice di un vettore.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	62h	BOUND r16,m32	-	13	10	7
									62h	BOUND r32,m64	-	-	10	7

BSF

Bit scan forward - Ricerca in avanti del primo bit di **DEST** che vale **1**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	x	-	-	-	0FBCh	BSF r16,r16/m16	-	-	10+3n	6,42/7,43
									0FBCh	BSF r32,r32/m32	-	-	10+3n	6,42/7,43

n = numero di scansioni effettuate.

BSR

Bit scan reverse - Ricerca all'indietro del primo bit di **DEST** che vale **1**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	x	-	-	-	0FBDh	BSR r16,r16/m16	-	-	10+3n	6,103/7,104
									0FBDh	BSR r32,r32/m32	-	-	10+3n	6,103/7,104

n = numero di scansioni effettuate.

BSWAP

Byte swap - Scambio di byte.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	0FC8h	BSWAP r32	-	-	-	1

BT

Bit test - Test di un bit.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	x	0FA3h	BT r16/m16,r16	-	-	3/12	3/8
									0FA3h	BT r32/m32,r32	-	-	3/12	3/8
									0FBAh	BT r16/m16,i8	-	-	3/6	3/3
									0FBAh	BT r32/m32,i8	-	-	3/6	3/3

BTC

Bit test and complement - Test e inversione di un bit.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	x	0FBBh	BTC r16/m16,r16	-	-	6/13	6/13
									0FBBh	BTC r32/m32,r32	-	-	6/13	6/13
									0FBAh	BTC r16/m16,i8	-	-	6/8	6/8
									0FBAh	BTC r32/m32,i8	-	-	6/8	6/8

BTR

Bit test and reset - Test e azzeramento di un bit.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	x	0FB3h	BTR r16/m16,r16	-	-	6/13	6/13
									0FB3h	BTR r32/m32,r32	-	-	6/13	6/13
									0FBAh	BTR r16/m16,i8	-	-	6/8	6/8
									0FBAh	BTR r32/m32,i8	-	-	6/8	6/8

BTS

Bit test and set - Test e attivazione di un bit.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	x	0FABh	BTS r16/m16,r16	-	-	6/13	6/13
									0FABh	BTS r32/m32,r32	-	-	6/13	6/13
									0FBAh	BTS r16/m16,i8	-	-	6/8	6/8
									0FBAh	BTS r32/m32,i8	-	-	6/8	6/8

C

CALL

Call procedure - Chiamata di un sottoprogramma.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	E8h	CALL rel16	19	7	7	3
									FFh	CALL r16/m16	16/21+EA	7/11	7/10	5/5
									9Ah	CALL p16:16	28	13	17	18
									FFh	CALL m16:m16	37+EA	16/29	22	17
									E8h	CALL rel32	-	-	7	3
									FFh	CALL r32/m32	-	-	7/10	5/5
									9Ah	CALL p16:32	-	-	17	18
									FFh	CALL m16:m32	-	-	22	17

CBW

Convert byte to word - Converte un **BYTE** in una **WORD** attraverso l'estensione del bit di segno.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	98h	CBW	2	2	3	3

CDQ

Convert doubleword to quadword - Converte una **DWORD** in una **QWORD** attraverso l'estensione del bit di segno.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	99h	CDQ	-	-	2	3

CLC

Clear carry flag - Pone **CF = 0**

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	0	F8h	CLC	2	2	2	2

CLD

Clear direction flag - Pone **DF = 0**

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	0	-	-	-	-	-	-	-	FCh	CLD	2	2	2	2

CLI

Clear interrupt enable flag - Pone **IF = 0**

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	0	-	-	-	-	-	-	FAh	CLI	2	2	3	5

CMC

Complement carry flag - Inverte il contenuto di **CF**

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	x	F5h	CMC	2	2	2	2

CMP

Compare two operands - Comparazione tra due operandi.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	3Ch	CMP AL,i8	4	3	2	1
									3Dh	CMP AX,i16	4	3	2	1
									3Dh	CMP EAX,i32	-	-	2	1
									80h	CMP r8/m8,i8	4/10+EA	3/6	2/5	1/2
									81h	CMP r16/m16,i16	4/10+EA	3/6	2/5	1/2
									81h	CMP r32/m32,i32	-	-	2/5	1/2
									83h	CMP r16/m16,i8	4/10+EA	3/6	2/5	1/2
									83h	CMP r32/m32,i8	-	-	2/5	1/2
									38h	CMP r8/m8,r8	3/9+EA	2/7	2/5	1/2
									39h	CMP r16/m16,r16	3/9+EA	2/7	2/5	1/2
									39h	CMP r32/m32,m32	-	-	2/5	1/2
									3Ah	CMP r8,r8/m8	3/9+EA	2/6	2/6	1/2
									3Bh	CMP r16,r16/m16	3/9+EA	2/6	2/6	1/2
									3Bh	CMP r32,r32/m32	-	-	2/6	1/2

CMPS, CMPSB, CMPSW, CMPSD

Compare string data - Comparazione tra stringhe.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	A6h	CMPS m8,m8	22	8	10	8
									A7h	CMPS m16,m16	22	8	10	8
									A7h	CMPS m32,m32	-	-	10	8
									A6h	CMPSB	22	8	10	8
									A7h	CMPSW	22	8	10	8
									A7h	CMPSD	-	-	10	8

CMPXCHG

Compare and exchange - Compara e scambia.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	0FB0h	CMPXCHG r8/m8,r8	-	-	-	6/7,6/10
									0FB1h	CMPXCHG r16/m16,r16	-	-	-	6/7,6/10
									0FB1h	CMPXCHG r32/m32,r32	-	-	-	6/7,6/10

CMPXCHG8B

Compare and exchange 8 byte - Compara e scambia due operandi a 64 bit.

Flags									Opcode	Istruzione	Clocks
O	D	I	T	S	Z	A	P	C			80586
-	-	-	-	-	x	-	-	-	0FC7h	CMPXCHG8B r64/m64	10

r64 si riferisce ad un registro della **FPU**.

CPUID

CPU identification - Identificazione del modello di CPU installato nel computer.

Flags									Opcode	Istruzione	Clocks
O	D	I	T	S	Z	A	P	C			80586
-	-	-	-	-	-	-	-	-	0FA2h	CPUID	14

CWD, CWDE

Convert word to doubleword - Converte una **WORD** in una **DWORD** attraverso l'estensione del bit di segno.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	99h	CWD	5	2	2	3
									98h	CWDE	-	-	3	3

D

DAA

Decimal adjust AL after addition – Aggiustamento del contenuto di **AL** dopo una addizione tra due numeri **Packed BCD** a **8** bit.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
?	-	-	-	x	x	x	x	x	27h	DAA	4	3	4	2

DAS

Decimal adjust AL after subtraction – aggiustamento del contenuto di **AL** dopo una sottrazione tra due numeri **Packed BCD** a **8** bit.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
?	-	-	-	x	x	x	x	x	2Fh	DAS	4	3	4	2

DEC

Decrement by 1 – Esegue la sottrazione:
DEST = DEST - 1.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	-	FEh	DEC r8/m8	3/15+EA	2/7	2/6	1/3
									FFh	DEC r16/m16	3/15+EA	2/7	2/6	1/3
									FFh	DEC r32/m32	-	-	2/6	1/3
									48h	DEC r16	3	2	2	1
									48h	DEC r32	-	-	2	1

DIV

Unsigned division of AL/AX/EAX – Divisione di **AL/AX/EAX**, per un numero intero senza segno.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
?	-	-	-	?	?	?	?	?	F6h	DIV r8/m8	80/60+EA	14/17	14/17	16/16
									F7h	DIV r16/m16	144,154+EA	22/25	22/25	24/24
									F7h	DIV r32/m32	-	-	38/41	40/40

E

ENTER

High-level procedure entry - Predispone **SP/ESP** e **BP/EBP** per la gestione dei parametri e delle variabili locali delle procedure.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	C8h	ENTER i16,0	-	11	10	14
									C8h	ENTER i16,1	-	15	12	17
									C8h	ENTER i16,i8	-	12+4(n-1)	15+4(n-1)	17+3n

I

IDIV

Integer signed divide - Divisione tra numeri interi con segno.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
?	-	-	-	?	?	?	?	?	F6h	IDIV r8/m8	101,112/ 107,118+EA	17/20	19	19/20
									F7h	IDIV r16/m16	165,184/ 171,190+EA	25/28	27	27,28
									F7h	IDIV r32/m32	-	-	43	43,44

IMUL

Integer signed multiply - Moltiplicazione tra numeri interi con segno.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
x	-	-	-	?	?	?	?	x

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
F6h	IMUL r8/m8	80,98/ 86,104+EA	13/16	9,14/ 12,17	13,18/ 13,18
F7h	IMUL r16/m16	128,154/ 134,160+EA	21/24	9,22/ 12,25	13,26/ 13,26
F7h	IMUL r32/m32	-	-	9,38/ 12,41	12,42/ 13,42
0FAFh	IMUL r16,r16/m16	-	-	9,22/ 12,25	13,26/ 13,26
0FAFh	IMUL r32,r32/m32	-	-	9,38/ 12,41	13,42/ 13,42
6Bh	IMUL r16,r16/m16,i8	-	21/24	9,14/ 12,27	13,26/ 13,26
6Bh	IMUL r32,r32/m32,i8	-	-	9,14/ 12,17	13,42
6Bh	IMUL r16,i8	-	21/24	9,14/ 12,17	13,26
6Bh	IMUL r32,i8	-	-	9,14/ 12,17	13,42
69h	IMUL r16,r16/m16,i16	-	21/24	9,22/ 12,25	13,26/ 13,26
69h	IMUL r32,r32/m32,i32	-	-	9,38/ 12,41	13,42/ 13,42
69h	IMUL r16,i16	-	-	9,22/ 12,25	13,26/ 13,26
69h	IMUL r32,i32	-	-	9,38/ 12,41	13,42/ 13,42

IN

Input from port - lettura di **8/16/32** bit di dati da una porta hardware.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
E4h	IN AL,i8	10	5	12	14
E5h	IN AX,i8	10	5	12	14
E5h	IN EAX,i8	-	-	12	14
ECh	IN AL,DX	8	5	13	14
EDh	IN AX,DX	8	5	13	14
EDh	IN EAX,DX	-	-	13	14

INC

Increment by 1 - Esegue la somma:
DEST = DEST + 1.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	-	FEh	INC r8/m8	3/15+EA	2/7	2/6	1/3
									FFh	INC r16/m16	3/15+EA	2/7	2/6	1/3
									FFh	INC r32/m32	-	-	2/6	1/3
									40h	INC r16	3	2	2	1
									40h	INC r32	-	-	2	1

INS, INSB, INSW, INSD

Input from port to string - Trasferimento dati da una porta hardware ad una stringa.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	E4h	INS m8,DX	-	5	15	17
									E5h	INS m16,DX	-	5	15	17
									E5h	INS m32,DX	-	-	15	17
									E4h	INSB	-	5	15	17
									E5h	INSW	-	5	15	17
									E5h	INSD	-	-	15	17

INT, INTO

Call to interrupt procedure - chiamata di un gestore di interruzione.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	0	0	-	-	-	-	-	CCh	INT3	52	23	33	26
									CCh	INT3	-	40	-	44
									CCh	INT3	-	78	-	71
									CDh	INT i8	51	23	37	30
									CEh	INTO	4/53	3/24	3/35	3/28

IRET

Interrupt return - ritorno da un gestore di interruzione.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	x	x	x	x	x	x	x	x	CFh	IRET	32	17	22	15

J

Jcond

Jump if condition is met - salta se la condizione e' verificata.

Flags								
O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks			
		8086	80286	80386	80486
75h	JO rel8	16,4	7,3	7,3	3,1
78h	JS rel8	16,4	7,3	7,3	3,1
7Ah	JP/JPE rel8	16,4	7,3	7,3	3,1
72h	JC rel8	16,4	7,3	7,3	3,1
74h	JE/JZ rel8	16,4	7,3	7,3	3,1
71h	JNO rel8	16,4	7,3	7,3	3,1
79h	JNS rel8	16,4	7,3	7,3	3,1
7Bh	JNP/JPO rel8	16,4	7,3	7,3	3,1
73h	JNC rel8	16,4	7,3	7,3	3,1
75h	JNE/JNZ rel8	16,4	7,3	7,3	3,1
77h	JA/JNBE rel8	16,4	7,3	7,3	3,1
73h	JAE/JNB rel8	16,4	7,3	7,3	3,1
72h	JB/JNAE rel8	16,4	7,3	7,3	3,1
76h	JBE/JNA rel8	16,4	7,3	7,3	3,1
7Fh	JG/JNLE rel8	16,4	7,3	7,3	3,1
7Dh	JGE/JNL rel8	16,4	7,3	7,3	3,1
7Ch	JL/JNGE rel8	16,4	7,3	7,3	3,1
7Eh	JLE/JNG rel8	16,4	7,3	7,3	3,1
0F80h	JO rel16/32	-	-	7,3	3,1
0F88h	JS rel16/32	-	-	7,3	3,1
0F8Ah	JP/JPE rel16/32	-	-	7,3	3,1
0F82h	JC rel16/32	-	-	7,3	3,1
0F84h	JE/JZ rel16/32	-	-	7,3	3,1
0F81h	JNO rel16/32	-	-	7,3	3,1
0F89h	JNS rel16/32	-	-	7,3	3,1
0F8Bh	JNP/JPO rel16/32	-	-	7,3	3,1
0F83h	JNC rel16/32	-	-	7,3	3,1
0F85h	JNE/JNZ rel16/32	-	-	7,3	3,1
0F87h	JA/JNBE rel16/32	-	-	7,3	3,1
0F83h	JAE/JNB rel16/32	-	-	7,3	3,1
0F82h	JB/JNAE rel16/32	-	-	7,3	3,1
0F86h	JBE/JNA rel16/32	-	-	7,3	3,1
0F8Fh	JG/JNLE rel16/32	-	-	7,3	3,1
0F8Dh	JGE/JNL rel16/32	-	-	7,3	3,1
0F8Ch	JL/JNGE rel16/32	-	-	7,3	3,1
0F8Eh	JLE/JNG rel16/32	-	-	7,3	3,1

JCXZ, JECXZ

Jump short if CX/ECX is zero - salto corto se **CX/ECX=0**.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	E3h	JCXZ rel8	18,6	8,4	9	3,1
									E3h	JECXZ rel8	-	-	9	3,1

JMP

Unconditional jump - salto incondizionato.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	EBh	JMP rel8	15	7	7	3
									E9h	JMP rel16	15	7	7	3
									FFh	JMP r16/m16	11/18+EA	7/11	7/10	5/5
									EAh	JMP p16:16	15	11	12	17
									FFh	JMP m16:m16	-	15	43	13
									E9h	JMP rel32	-	-	7	3
									FFh	JMP r32/m32	-	-	7	5/5
									EAh	JMP p16:32	-	-	12	13
									FFh	JMP m16:m32	-	-	43	13

L

LAHF

Load status flags into AH register - Trasferisce in **AH** gli **8** bit meno significativi del registro **FLAGS**.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	9Fh	LAHF	4	2	2	3

LDS, LES, LFS, LGS, LSS

Load far pointer - Trasferisce un indirizzo logico **Seg:Offset** nella coppia **sreg:reg** specificata dal mnemonico dell'istruzione stessa e dall'operando **DEST**.
LFS, LGS, LSS Solo CPU 80386 e superiori.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	C5h	LDS r16,m16:16	16+EA	7	7	6/12
									C5h	LDS r32,m16:32	-	-	7	6/12
									C4h	LES r16,m16:16	16+EA	7	7	6/12
									C4h	LES r32,m16:32	-	-	7	6/12
									0FB4h	LFS r16,m16:16	-	-	7	6/12
									0FB4h	LFS r32,m16:32	-	-	7	6/12
									0FB5h	LGS r16,m16:16	-	-	7	6/12
									0FB5h	LGS r32,m16:32	-	-	7	6/12
									0FB2h	LSS r16,m16:16	-	-	7	6/12
									0FB2h	LSS r32,m16:32	-	-	7	6/12

LEA

Load effective address - Trasferisce nel registro **DEST**, l'**effective address** specificato dall'operando **SRC**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	8Dh	LEA r16,m16	2+EA	3	2	1
									8Dh	LEA r32,m16	-	-	2	1
									8Dh	LEA r16,m32	-	-	2	1
									8Dh	LEA r32,m32	-	-	2	1

LEAVE

High-level procedure exit - Ripristina **SP/ESP** e **BP/EBP** alla fine di una procedura dotata di parametri e variabili locali.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	C9h	LEAVE	-	5	4	5
									C9h	LEAVE	-	-	4	5

LODS, LODSB, LODSW, LODSD

Load string data - Accesso in lettura ad una stringa.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	ACh	LODS m8	12	5	5	5
									ADh	LODS m16	12	5	5	5
									ADh	LODS m32	-	-	5	5
									ACh	LODSB	12	5	5	5
									ADh	LODSW	12	5	5	5
									ADh	LODSD	-	-	5	5

LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ

Loop control with CX register - controllo di una iterazione attraverso il registro **CX**

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	E2h	LOOP rel8	17	8	11	2,6
									E1h	LOOPE rel8	18	8	11	9,6
									E1h	LOOPZ rel8	18	8	11	9,6
									E0h	LOOPNE rel8	19	8	11	9,6
									E0h	LOOPNZ rel8	19	8	11	9,6

M

MOV

Move data - Copia, bit per bit, da **SRC** a **DEST**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	88h	MOV r8/m8,r8	2/9+EA	2/3	2/2	1
									89h	MOV r16/m16,r16	2/9+EA	2/3	2/2	1
									89h	MOV r32/m32,r32	-	-	2/2	1
									8Ah	MOV r8,r8/m8	2/8+EA	2/5	2/4	1
									8Bh	MOV r16,r16/m16	2/8+EA	2/5	2/4	1
									8Bh	MOV r32,r32/m32	-	-	2/4	1
									8Ch	MOV r16/m16,sr	2/9+EA	2/3	2/2	3/3
									8Dh	MOV sr,r16/m16	2/8+EA	2/5	2/5	3/9
									A0h	MOV AL,o8	10	5	4	1
									A1h	MOV AX,o16	10	5	4	1
									A1h	MOV EAX,o32	-	-	4	1
									A2h	MOV o8,AL	10	3	4	1
									A3h	MOV o16,AX	10	3	2	1
									A3h	MOV o32,EAX	-	-	2	1
									B0h	MOV r8,i8	4	2	2	1
									B8h	MOV r16,i16	4	2	2	1
									B8h	MOV r32,i32	-	-	2	1
									C6h	MOV r8/m8,i8	4/10+EA	2/3	2/2	1
									C7h	MOV r16/m16,i16	4/10+EA	2/3	2/2	1
									C7h	MOV r32/m32,i32	-	-	2/2	1

MOVS, MOVSB, MOVSW, MOVSD

Move data from string to string - Trasferimento dati da stringa a stringa.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	A4h	MOVS m8,m8	18	5	7	7
									A5h	MOVS m16,m16	18	5	7	7
									A5h	MOVS m32,m32	-	-	7	7
									A4h	MOVSB	18	5	7	7
									A5h	MOVSW	18	5	7	7
									A5h	MOVSD	-	-	7	7

MOVSX

Move with sign-extension - Copia, bit per bit, da **SRC** a **DEST** con estensione del bit di segno.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
0FBEh	MOVSX r16,r8/m8	-	-	3/3	3/6
0FBEh	MOVSX r32,r8/m8	-	-	3/3	3/6
0FBFh	MOVSX r32,r16/m16	-	-	3/3	3/6

MOVZX

Move with zero-extension - Copia, bit per bit, da **SRC** a **DEST** con estensione di zeri.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
0FB6h	MOVZX r16,r8/m8	-	-	3/3	3/6
0FB6h	MOVZX r32,r8/m8	-	-	3/3	3/6
0FB7h	MOVZX r32,r16/m16	-	-	3/3	3/6

MUL

Unsigned multiplication of AL/AX/EAX - Moltiplicazione di un numero intero senza segno, per **AL/AX/EAX**.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
x	-	-	-	?	?	?	?	x

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
F6h	MUL r8/m8	70,77/ 76,83+EA	13/16	9,14/ 12,17	13/18
F7h	MUL r16/m16	118,113/ 124,139+EA	21/24	9,22/ 12,25	13,26/ 13,26
F7h	MUL r32/m32	-	-	9,38/ 12,41	13,42/ 13,42

N

NEG

Two's complement negation - Calcola il complemento a **2** di **DEST** (cambiamento di segno).

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	F6h	NEG r8/m8	3/16+EA	2/7	2/6	1/3
									F7h	NEG r16/m16	3/16+EA	2/7	2/6	1/3
									F7h	NEG r32/m32	-	-	2/6	1/3

NOP

No operation - Nessuna operazione.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	90h	NOP	3	3	3	1

NOT

One's complement negation - Inverte tutti i bit di **DEST** (complemento a uno).

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	F6h	NOT r8/m8	3/16+EA	2/7	2/6	1/3
									F7h	NOT r16/m16	3/16+EA	2/7	2/6	1/3
									F7h	NOT r32/m32	-	-	2/6	1/3

O

OR

Logical inclusive OR - Esegue un **OR** logico inclusivo tra **SRC** e **DEST**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
0	-	-	-	x	x	?	x	0	0Ch	OR AL,i8	4	3	2	1
									0Dh	OR AX,i16	4	3	2	1
									0Dh	OR EAX,i32	-	-	2	1
									80h	OR r8/m8,i8	4/17+EA	3/7	2/7	1/3
									81h	OR r16/m16,i16	4/17+EA	3/7	2/7	1/3
									81h	OR r32/m32,i32	-	-	2/7	1/3
									83h	OR r16/m16,i8	4/17+EA	3/7	2/7	1/3
									83h	OR r32/m32,i8	-	-	2/7	1/3
									08h	OR r8/m8,r8	3/16+EA	2/7	2/6	1/3
									09h	OR r16/m16,r16	3/16+EA	2/7	2/6	1/3
									09h	OR r32/m32,r32	-	-	2/6	1/3
									0Ah	OR r8,r8/m8	3/9+EA	2/7	2/7	1/2
									0Bh	OR r16,r16/m16	3/9+EA	2/7	2/7	1/2
									0Bh	OR r32,r32/m32	-	-	2/7	1/2

OUT

Output to port - scrittura di **8/16/32** bit di dati in una porta hardware.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	E6h	OUT i8,AL	10	3	10	16
									E7h	OUT i8,AX	10	3	10	16
									E7h	OUT i8,EAX	-	-	10	16
									EEh	OUT DX,AL	8	3	11	16
									EFh	OUT DX,AX	8	3	11	16
									EFh	OUT DX,EAX	-	-	11	16

OUTS, OUTSB, OUTSW, OUTSD

Output from string to port - Trasferimento dati da una stringa ad una porta hardware.

Flags								
O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks			
		8086	80286	80386	80486
6Eh	OUTS DX,m8	-	5	14	17
6Fh	OUTS DX,m16	-	5	14	17
6Fh	OUTS DX,m32	-	-	14	17
6Eh	OUTSB	-	5	14	17
6Fh	OUTSW	-	5	14	17
6Fh	OUTSD	-	-	14	17

P

POP

Pop a value from the stack - Estrazione di un valore dalla cima dello stack.

Flags								
O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks			
		8086	80286	80386	80486
8Fh	POP m16	17+EA	5	5	6
8Fh	POP m32	-	-	5	6
58h	POP r16	8	5	4	4
58h	POP r32	-	-	4	4
1Fh	POP DS	8	5	7	3
07h	POP ES	8	5	7	3
17h	POP SS	8	5	7	3
0FA1h	POP FS	-	-	7	3
0FA9h	POP GS	-	-	7	3

POPA, POPAD

Pop all general registers from the stack - Estrazione dalla cima dello stack, di valori da trasferire in tutti i registri generali.

Flags								
O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks			
		8086	80286	80386	80486
61h	POPA	-	19	24	9
61h	POPAD	-	-	24	9

POPF, POPFD

Pop from stack into FLAGS/EFLAGS register - Estrazione dalla cima dello stack, di **16/32** bit da trasferire nel registro **FLAGS/EFLAGS**.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
x	x	x	x	x	x	x	x	x	9Dh	POPF	8	5	5	9
									9Dh	POPFD	-	-	5	9

PUSH

Push operand onto the stack - Inserimento di un operando sulla cima dello stack.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	FFh	PUSH m16	16+EA	5	5	4
									FFh	PUSH m32	-	-	5	4
									50h	PUSH r16	11	3	2	1
									50h	PUSH r32	-	-	2	1
									6Ah	PUSH i8	-	3	2	1
									68h	PUSH i16	-	3	2	1
									68h	PUSH i32	-	-	2	1
									0Eh	PUSH CS	10	3	2	3
									1Eh	PUSH DS	10	3	2	3
									06h	PUSH ES	10	3	2	3
									16h	PUSH SS	10	3	2	3
									0FA0h	PUSH FS	-	-	2	3
									0FA8h	PUSH GS	-	-	2	3

PUSHA, PUSHAD

Push all general registers onto the stack - Inserimento sulla cima dello stack, del contenuto di tutti i registri generali.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	60h	PUSHA	-	17	18	11
									60h	PUSHAD	-	-	18	11

PUSHF, PUSHFD

Push FLAGS/EFLAGS register onto the stack - Inserimento sulla cima dello stack, del contenuto del registro **FLAGS/EFLAGS**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	9Ch	PUSHF	10	3	4	4
									9Ch	PUSHFD	-	-	4	4

R

RCL

Rotate through carry left - Rotazione dei bit di **DEST** verso sinistra attraverso **CF**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	?	x	x	D0h	RCL r8/m8,1	2/15+EA	2/7	9/10	3/4
									D2h	RCL r8/m8,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
									C0h	RCL r8/m8,i8 (*)	-	5/8	9/10	8,30/ 9,31
									D1h	RCL r16/m16,1	2/15+EA	2/7	9/10	3/4
									D3h	RCL r16/m16,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
									C1h	RCL r16/m16,i8 (*)	-	5/8	9/10	8,30/ 9,31
									D1h	RCL r32/m32,1	-	-	9/10	3/4
									D3h	RCL r32/m32,CL	-	-	9/10	8,30/ 9,31
									C1h	RCL r32/m32,i8	-	-	9/10	8,30/ 9,31

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni rotazione da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni rotazione da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

RCR

Rotate through carry right - Rotazione dei bit di **DEST** verso destra attraverso **CF**.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
x	-	-	-	x	x	?	x	x

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
D0h	RCR r8/m8,1	2/15+EA	2/7	9/10	3/4
D2h	RCR r8/m8,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
C0h	RCR r8/m8,i8 (*)	-	5/8	9/10	8,30/ 9,31
D1h	RCR r16/m16,1	2/15+EA	2/7	9/10	3/4
D3h	RCR r16/m16,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
C1h	RCR r16/m16,i8 (*)	-	5/8	9/10	8,30/ 9,31
D1h	RCR r32/m32,1	-	-	9/10	3/4
D3h	RCR r32/m32,CL	-	-	9/10	8,30/ 9,31
C1h	RCR r32/m32,i8	-	-	9/10	8,30/ 9,31

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni rotazione da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni rotazione da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

REP, REPE/REPZ, REPNE/REPNZ

Repeat following string operation - Iterazione di un'istruzione per le stringhe.

O	D	I	T	S	Z	A	P	C
-	-	-	-	-	-	-	-	-

Opcode	Istruzione	Clocks 8086	80286	80386	80486
F36Ch	REP INS m8,DX	-	5+4CX	27+6CX	30+8CX
F36Dh	REP INS m16,DX	-	5+4CX	27+6CX	30+8CX
F36Dh	REP INS m32,DX	-	-	27+6CX	30+8CX
F3A4h	REP MOVS m8,m8	9+17CX	5+4CX	5+4CX	12+3CX
F3A5h	REP MOVS m16,m16	9+17CX	5+4CX	5+4CX	12+3CX
F3A5h	REP MOVS m32,m32	-	-	5+4CX	12+3CX
F36Eh	REP OUTS DX,m8	-	5+4CX	26+5CX	31+5CX
F36Fh	REP OUTS DX,m16	-	5+4CX	26+5CX	31+5CX
F36Fh	REP OUTS DX,m32	-	-	26+5CX	31+5CX
F3ACh	REP LODS m8	9+10CX	4+3CX	5+5CX	7+4CX
F3ADh	REP LODS m16	9+10CX	4+3CX	5+5CX	7+4CX
F3ADh	REP LODS m32	-	-	5+5CX	7+4CX
F3AAh	REP STOS m8	9+10CX	4+3CX	5+5CX	7+4CX
F3ABh	REP STOS m16	9+10CX	4+3CX	5+5CX	7+4CX
F3ABh	REP STOS m32	-	-	5+5CX	7+4CX
F3A6h	REPE CMPS m8,m8	9+22N	5+9N	5+9N	7+7CX
F3A7h	REPE CMPS m16,m16	9+22N	5+9N	5+9N	7+7CX
F3A7h	REPE CMPS m32,m32	-	-	5+9N	7+7CX
F3AEh	REPE SCAS m8	9+15N	5+8N	5+8N	7+5CX
F3AFh	REPE SCAS m16	9+15N	5+8N	5+8N	7+5CX
F3AFh	REPE SCAS m32	-	-	5+8N	7+5CX
F2A6h	REPNE CMPS m8,m8	9+22N	5+9N	5+9N	7+7CX
F2A7h	REPNE CMPS m16,m16	9+22N	5+9N	5+9N	7+7CX
F2A7h	REPNE CMPS m32,m32	-	-	5+9N	7+7CX
F2AEh	REPNE SCAS m8	9+15N	5+8N	5+8N	7+5CX
F2AFh	REPNE SCAS m16	9+15N	5+8N	5+8N	7+5CX
F2AFh	REPNE SCAS m32	-	-	5+8N	7+5CX

N = numero di iterazioni.

RET

Return from procedure - ritorno da un sottoprogramma.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	C3h	RET	16	11	10	5
									CBh	RET	26	15	18	13
									CBh	RET	-	55	-	13
									C2h	RET i16	20	11	10	5
									CAh	RET i16	25	15	18	14
									CAh	RET i16	-	55	-	14

ROL

Rotate left - Rotazione dei bit di **DEST** verso sinistra.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	?	x	x	D0h	ROL r8/m8,1	2/15+EA	2/7	9/10	3/4
									D2h	ROL r8/m8,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
									C0h	ROL r8/m8,i8 (*)	-	5/8	9/10	8,30/ 9,31
									D1h	ROL r16/m16,1	2/15+EA	2/7	9/10	3/4
									D3h	ROL r16/m16,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
									C1h	ROL r16/m16,i8 (*)	-	5/8	9/10	8,30/ 9,31
									D1h	ROL r32/m32,1	-	-	9/10	3/4
									D3h	ROL r32/m32,CL	-	-	9/10	8,30/ 9,31
									C1h	ROL r32/m32,i8	-	-	9/10	8,30/ 9,31

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni rotazione da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni rotazione da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

ROR

Rotate right - Rotazione dei bit di **DEST** verso destra.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	?	x	x	D0h	ROR r8/m8,1	2/15+EA	2/7	9/10	3/4
									D2h	ROR r8/m8,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
									C0h	ROR r8/m8,i8 (*)	-	5/8	9/10	8,30/ 9,31
									D1h	ROR r16/m16,1	2/15+EA	2/7	9/10	3/4
									D3h	ROR r16/m16,CL (*)	8/20+EA	5/8	9/10	8,30/ 9,31
									C1h	ROR r16/m16,i8 (*)	-	5/8	9/10	8,30/ 9,31
									D1h	ROR r32/m32,1	-	-	9/10	3/4
									D3h	ROR r32/m32,CL	-	-	9/10	8,30/ 9,31
									C1h	ROR r32/m32,i8	-	-	9/10	8,30/ 9,31

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni rotazione da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni rotazione da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

S

SAHF

Store AH into flags - Trasferisce il contenuto di **AH** negli **8** bit meno significativi del registro **FLAGS**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	x	x	x	x	x	9Eh	SAHF	4	2	3	2

SAL

Shift arithmetic left - Scorrimento aritmetico dei bit di **DEST** verso sinistra.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
x	-	-	-	x	x	?	x	x

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
D0h	SAL r8/m8,1	2/15+EA	2/7	3/7	3/4
D2h	SAL r8/m8,CL (*)	8/20+EA	5/8	3/7	3/4
C0h	SAL r8/m8,i8 (*)	-	5/8	3/7	2/4
D1h	SAL r16/m16,1	2/15+EA	2/7	3/7	3/4
D3h	SAL r16/m16,CL (*)	8/20+EA	5/8	3/7	3/4
C1h	SAL r16/m16,i8 (*)	-	5/8	3/7	2/4
D1h	SAL r32/m32,1	-	-	3/4	3/7
D3h	SAL r32/m32,CL	-	-	3/4	3/7
C1h	SAL r32/m32,i8	-	-	2/4	3/7

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni scorrimento da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni scorrimento da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

SAR

Shift arithmetic right - Scorrimento aritmetico dei bit di **DEST** verso destra.

Flags								
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**
0	-	-	-	x	x	?	x	x

Opcode	Istruzione	Clocks			
		8086	**80286**	**80386**	**80486**
D0h	SAR r8/m8,1	2/15+EA	2/7	3/7	3/4
D2h	SAR r8/m8,CL (*)	8/20+EA	5/8	3/7	3/4
C0h	SAR r8/m8,i8 (*)	-	5/8	3/7	2/4
D1h	SAR r16/m16,1	2/15+EA	2/7	3/7	3/4
D3h	SAR r16/m16,CL (*)	8/20+EA	5/8	3/7	3/4
C1h	SAR r16/m16,i8 (*)	-	5/8	3/7	2/4
D1h	SAR r32/m32,1	-	-	3/4	3/7
D3h	SAR r32/m32,CL	-	-	3/4	3/7
C1h	SAR r32/m32,i8	-	-	2/4	3/7

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni scorrimento da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni scorrimento da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

SBB

Subtraction with borrow - Esegue la sottrazione:
DEST = DEST - (SRC + CF).

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	1Ch	SBB AL,i8	4	3	2	1
									1Dh	SBB AX,i16	4	3	2	1
									1Dh	SBB EAX,i32	-	-	2	1
									80h	SBB r8/m8,i8	4/17+EA	3/7	2/7	1/3
									81h	SBB r16/m16,i16	4/17+EA	3/7	2/7	1/3
									81h	SBB r32/m32,i32	-	-	2/7	1/3
									83h	SBB r16/m16,i8	4/17+EA	3/7	2/7	1/3
									83h	SBB r32/m32,i8	-	-	2/7	1/3
									18h	SBB r8/m8,r8	3/16+EA	2/7	2/6	1/3
									19h	SBB r16/m16,r16	3/16+EA	2/7	2/6	1/3
									19h	SBB r32/m32,r32	-	-	2/6	1/3
									1Ah	SBB r8,r8/m8	3/9+EA	2/7	2/7	1/2
									1Bh	SBB r16,r16/m16	3/9+EA	2/7	2/7	1/2
									1Bh	SBB r32,r32/m32	-	-	2/7	1/2

SCAS, SCASB, SCASW, SCASD

Scan string data - Comparazione di un elemento di una stringa.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	AEh	SCAS m8	15	7	7	6
									AFh	SCAS m16	15	7	7	6
									AFh	SCAS m32	-	-	7	6
									AEh	SCASB	15	7	7	6
									AFh	SCASW	15	7	7	6
									AFh	SCASD	-	-	7	6

SETcond

Set byte if condition is met - abilita un byte se la condizione e' verificata.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	0F90h	SETO r8/m8	-	-	4/5	3/4
									0F98h	SETS r8/m8	-	-	4/5	3/4
									0F9Ah	SETP/SETPE r8/m8	-	-	4/5	3/4
									0F92h	SETC r8/m8	-	-	4/5	3/4
									0F94h	SETE/SETZ r8/m8	-	-	4/5	3/4
									0F91h	SETNO r8/m8	-	-	4/5	3/4
									0F99h	SETNS r8/m8	-	-	4/5	3/4
									0F9Bh	SETNP/SETPO r8/m8	-	-	4/5	3/4
									0F93h	SETNC r8/m8	-	-	4/5	3/4
									0F95h	SETNE/SETNZ r8/m8	-	-	4/5	3/4
									0F97h	SETA/SETNBE r8/m8	-	-	4/5	3/4
									0F93h	SETAE/SETNB r8/m8	-	-	4/5	3/4
									0F92h	SETB/SETNAE r8/m8	-	-	4/5	3/4
									0F96h	SETBE/SETNA r8/m8	-	-	4/5	3/4
									0F9Fh	SETG/SETNLE r8/m8	-	-	4/5	3/4
									0F9Dh	SETGE/SETNL r8/m8	-	-	4/5	3/4
									0F9Ch	SETL/SETNGE r8/m8	-	-	4/5	3/4
									0F9Eh	SETLE/SETNG r8/m8	-	-	4/5	3/4

SHL

Shift logical left - Scorrimento logico dei bit di **DEST** verso sinistra.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
x	-	-	-	x	x	?	x	x	D0h	SHL r8/m8,1	2/15+EA	2/7	3/7	3/4
									D2h	SHL r8/m8,CL (*)	8/20+EA	5/8	3/7	3/4
									C0h	SHL r8/m8,i8 (*)	-	5/8	3/7	2/4
									D1h	SHL r16/m16,1	2/15+EA	2/7	3/7	3/4
									D3h	SHL r16/m16,CL (*)	8/20+EA	5/8	3/7	3/4
									C1h	SHL r16/m16,i8 (*)	-	5/8	3/7	2/4
									D1h	SHL r32/m32,1	-	-	3/4	3/7
									D3h	SHL r32/m32,CL	-	-	3/4	3/7
									C1h	SHL r32/m32,i8	-	-	2/4	3/7

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni scorrimento da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni scorrimento da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

SHLD

Shift logical left, double precision - Scorrimento logico dei bit di **DEST** verso sinistra, in doppia precisione.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	?	x	x	0FA4h	SHLD r16/m16,r16,i8	-	-	3/7	2/3
									0FA4h	SHLD r32/m32,r32,i8	-	-	3/7	2/3
									0FA5h	SHLD r16/m16,r16,CL	-	-	3/7	3/4
									0FA5h	SHLD r32/m32,r32,CL	-	-	3/7	3/4

(il flag **OF** ha senso solo se il contatore vale **1**)

SHR

Shift logical right - Scorrimento logico dei bit di **DEST** verso destra.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	?	x	x	D0h	SHR r8/m8,1	2/15+EA	2/7	3/7	3/4
									D2h	SHR r8/m8,CL (*)	8/20+EA	5/8	3/7	3/4
									C0h	SHR r8/m8,i8 (*)	-	5/8	3/7	2/4
									D1h	SHR r16/m16,1	2/15+EA	2/7	3/7	3/4
									D3h	SHR r16/m16,CL (*)	8/20+EA	5/8	3/7	3/4
									C1h	SHR r16/m16,i8 (*)	-	5/8	3/7	2/4
									D1h	SHR r32/m32,1	-	-	3/4	3/7
									D3h	SHR r32/m32,CL	-	-	3/4	3/7
									C1h	SHR r32/m32,i8	-	-	2/4	3/7

(*) CPU **8086** - aggiungere **4** cicli di clock per ogni scorrimento da **1** bit
(*) CPU **80286** - aggiungere **1** ciclo di clock per ogni scorrimento da **1** bit
(il flag **OF** ha senso solo se il contatore vale **1**)

SHRD

Shift logical right, double precision - Scorrimento logico dei bit di **DEST** verso destra, in doppia precisione.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	?	x	x	0FACh	SHRD r16/m16,r16,i8	-	-	3/7	2/3
									0FACh	SHRD r32/m32,r32,i8	-	-	3/7	2/3
									0FADh	SHRD r16/m16,r16,CL	-	-	3/7	3/4
									0FADh	SHRD r32/m32,r32,CL	-	-	3/7	3/4

(il flag **OF** ha senso solo se il contatore vale **1**)

STC

Set carry flag – Pone **CF = 1**

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	1	F9h	STC	2	2	2	2

STD

Set direction flag – Pone **DF = 1**

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	1	-	-	-	-	-	-	-	FDh	STD	2	2	2	2

STI

Set interrupt enable flag – Pone **IF = 1**

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	1	-	-	-	-	-	-	FBh	STI	2	2	3	5

STOS, STOSB, STOSW, STOSD

Store string data – Accesso in scrittura ad una stringa.

Flags									Opcode	Istruzione	Clocks			
O	**D**	**I**	**T**	**S**	**Z**	**A**	**P**	**C**			**8086**	**80286**	**80386**	**80486**
-	-	-	-	-	-	-	-	-	AAh	STOS m8	11	3	4	5
									ABh	STOS m16	11	3	4	5
									ABh	STOS m32	-	-	4	5
									AAh	STOSB	11	3	4	5
									ABh	STOSW	11	3	4	5
									ABh	STOSD	-	-	4	5

SUB

Subtraction - Esegue la sottrazione:
DEST = DEST - SRC.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	2Ch	SUB AL,i8	4	3	2	1
									2Dh	SUB AX,i16	4	3	2	1
									2Dh	SUB EAX,i32	-	-	2	1
									80h	SUB r8/m8,i8	4/17+EA	3/7	2/7	1/3
									81h	SUB r16/m16,i16	4/17+EA	3/7	2/7	1/3
									81h	SUB r32/m32,i32	-	-	2/7	1/3
									83h	SUB r16/m16,i8	4/17+EA	3/7	2/7	1/3
									83h	SUB r32/m32,i8	-	-	2/7	1/3
									28h	SUB r8/m8,r8	3/16+EA	2/7	2/6	1/3
									29h	SUB r16/m16,r16	3/16+EA	2/7	2/6	1/3
									29h	SUB r32/m32,m32	-	-	2/6	1/3
									2Ah	SUB r8,r8/m8	3/9+EA	2/7	2/7	1/2
									2Bh	SUB r16,r16/m16	3/9+EA	2/7	2/7	1/2
									2Bh	SUB r32,r32/m32	-	-	2/7	1/2

T

TEST

Logical compare - Esegue una comparazione logica (**AND**) tra **SRC** e **DEST**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
0	-	-	-	x	x	?	x	0	A8h	TEST AL,i8	4	3	2	1
									A9h	TEST AX,i16	4	3	2	1
									A9h	TEST EAX,i32	-	-	2	1
									F6h	TEST r8/m8,i8	5/11+EA	3/6	2/5	1/2
									F7h	TEST r16/m16,i16	5/11+EA	3/6	2/5	1/2
									F7h	TEST r32/m32,i32	-	-	2/5	1/2
									84h	TEST r8/m8,r8	3/9+EA	2/6	2/5	1/2
									85h	TEST r16/m16,r16	3/9+EA	2/6	2/5	1/2
									85h	TEST r32/m32,r32	-	-	2/5	1/2

W

WAIT

WAIT until the BUSY# pin is high - attendere finche' il coprocessore matematico e' occupato.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	9Bh	WAIT	4+5n	3	6	1,3

n = numero cicli di attesa della CPU.

X

XADD

Exchange and add - Scambia e somma due operandi.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
x	-	-	-	x	x	x	x	x	0FC0h	XADD r8/m8,r8	-	-	-	3/4
									0FC1h	XADD r16/m16,r16	-	-	-	3/4
									0FC2h	XADD r32/m32,r32	-	-	-	3/4

XCHG

Exchange register/memory with register - Scambia il contenuto di **SRC** e **DEST**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	86h	XCHG r8/m8,r8	4/17+EA	3/5	3/5	3/5
									86h	XCHG r8,r8/m8	4/17+EA	3/5	3/5	3/5
									87h	XCHG r16/m16,r16	4/17+EA	3/5	3/5	3/5
									87h	XCHG r16,r16/m16	4/17+EA	3/5	3/5	3/5
									87h	XCHG r32/m32,r32	-	-	3/5	3/5
									87h	XCHG r32,r32/m32	-	-	3/5	3/5
									90h	XCHG AX,r16	3	3	3	3
									90h	XCHG r16,AX	3	3	3	3
									90h	XCHG EAX,r32	-	-	3	3
									90h	XCHG r32,EAX	-	-	3	3

XLAT, XLATB

Table look-up translation - Conversione attraverso una tabella di consultazione.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
-	-	-	-	-	-	-	-	-	D7h	XLAT m8	11	5	5	4
									D7h	XLATB	11	5	5	4

XOR

Logical exclusive OR - Esegue un **OR** logico esclusivo tra **SRC** e **DEST**.

Flags									Opcode	Istruzione	Clocks			
O	D	I	T	S	Z	A	P	C			8086	80286	80386	80486
0	-	-	-	x	x	?	x	0	34h	XOR AL,i8	4	3	2	1
									35h	XOR AX,i16	4	3	2	1
									35h	XOR EAX,i32	-	-	2	1
									80h	XOR r8/m8,i8	4/17+EA	3/7	2/7	1/3
									81h	XOR r16/m16,i16	4/17+EA	3/7	2/7	1/3
									81h	XOR r32/m32,i32	-	-	2/7	1/3
									83h	XOR r16/m16,i8	4/17+EA	3/7	2/7	1/3
									83h	XOR r32/m32,i8	-	-	2/7	1/3
									30h	XOR r8/m8,r8	3/16+EA	2/7	2/6	1/3
									31h	XOR r16/m16,r16	3/16+EA	2/7	2/6	1/3
									31h	XOR r32/m32,r32	-	-	2/6	1/3
									32h	XOR r8,r8/m8	3/9+EA	2/7	2/7	1/2
									33h	XOR r16,r16/m16	3/9+EA	2/7	2/7	1/2
									33h	XOR r32,r32/m32	-	-	2/7	1/2